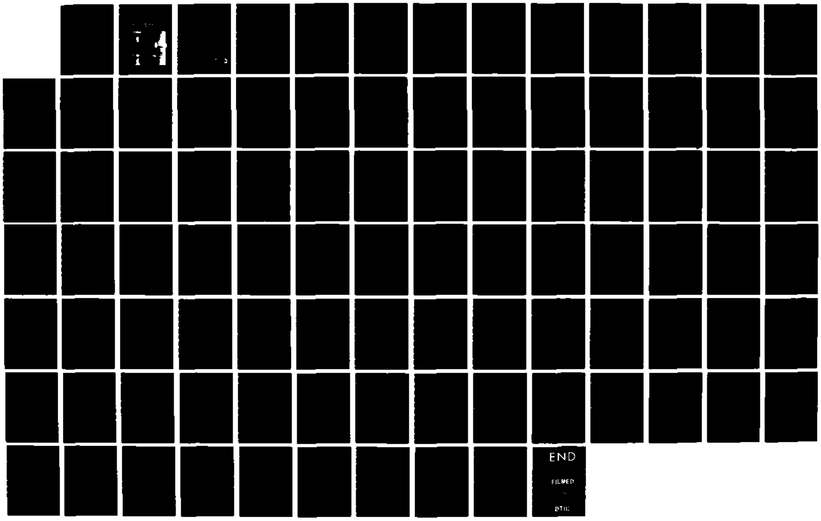
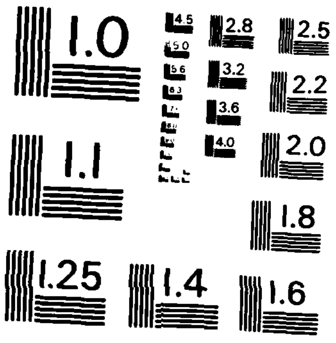
AD-A161 379   A SIMULATION PROGRAM WITH LATENCY EXPLOITATION AND NODE   1/1
TEARING(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE
LAB   T K YU FEB 85 R-1029 N00014-84-C-0149

UNCLASSIFIED   F/G 9/5   NL





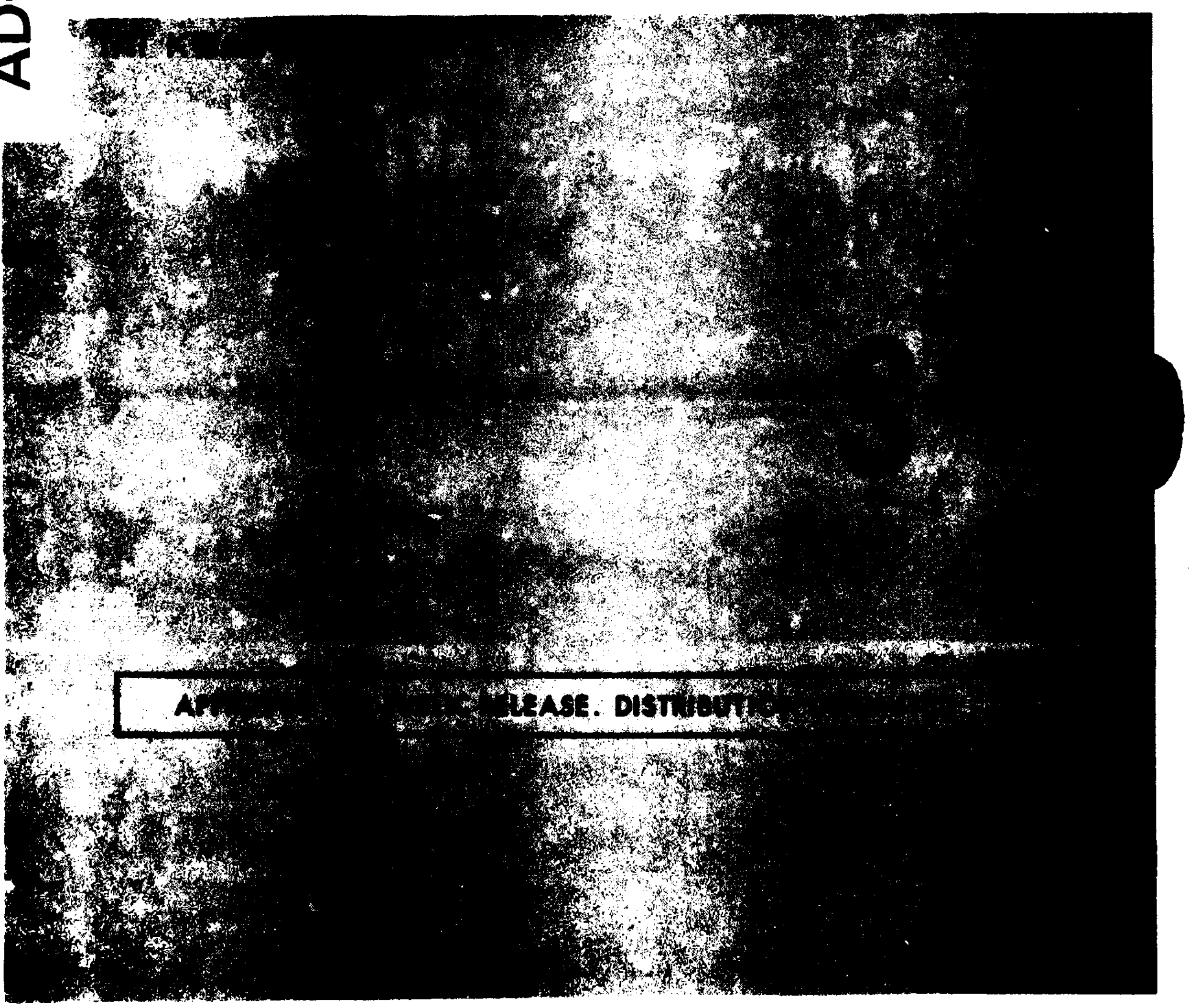
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

REPORT R-1029 FEBRUARY 1985 UILU-ENG 85-2204

(12)

COORDINATED [illegible] LABORATORY

# A SIMULATION PROGRAM WITH LATENCY EXPLOITATION AND NODE TEARING



UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

AD-A161 379

## REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Ur [illegible]assified | 1b. RESTRICTIVE MARKINGS<br>None |
| 2a. SECURITY CLASSIFICATION AUTHORITY<br>N/A | 3. DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for public release, distribution unlimited. |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE<br>N/A | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>R-report #1029 | 5. MONITORING ORGANIZATION REPORT NUMBER(S)<br>N/A |
| 6a. NAME OF PERFORMING ORGANIZATION<br>Coordinated Science Laboratory, Univ. of Illinois — 6b. OFFICE SYMBOL *(If applicable)* N/A | 7a. NAME OF MONITORING ORGANIZATION<br>Office of Naval Research |
| 6c. ADDRESS *(City, State and ZIP Code)*<br>1101 W. Springfield Avenue<br>Urbana, Illinois 61801 | 7b. ADDRESS *(City, State and ZIP Code)*<br>800 N. Quincy St.<br>Arlington, VA 22217 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION<br>ONR — 8b. OFFICE SYMBOL *(If applicable)* | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>Contract N00014- 84-C-0149 |
| 8c. ADDRESS *(City, State and ZIP Code)*<br>800 N. Quincy St.<br>Arlington, VA 22217 | 10. SOURCE OF FUNDING NOS. |

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT NO. |
|---|---|---|---|
| N/A | N/A | N/A | N/A |

11. TITLE *(Include Security Classification)* A Simulation Program with Latency Exploitation and Node Tearing

12. PERSONAL AUTHOR(S) Yu, Tat Kwan E.

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT *(Yr., Mo., Day)* | 15. PAGE COUNT |
|---|---|---|---|
| Technical Report | FROM ____ TO ____ | February, 1985 | 85 |

16. SUPPLEMENTARY NOTATION
N/A [illegible]

| 17. COSATI CODES | | |
|---|---|---|
| FIELD | GROUP | SUB. GR. |
| | | |
| | | |

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
circuit, subcircuit, ~~"standard" circuit simulator~~, SLATE, SPICE2, node tearing.

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Circuit simulation has become an indispensable tool in the design of integrated circuits. Standard circuit simulators, such as SPICE [1], can predict accurately the circuit performance. However, the use of these simulators is limited to circuits of several hundred transistors. As the size and complexity of the integrated circuit increase, the memory and cpu time requirements for such an analysis become prohibitive.

| | |
|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT<br>UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT. ☐ DTIC USERS ☐ | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE NUMBER *(Include Area Code)* — 22c. OFFICE SYMBOL None |

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

# A SIMULATION PROGRAM WITH LATENCY EXPLOITATION AND NODE TEARING

BY

TAT KWAN EDGAR YU

B.S., University of Michigan, Ann Arbor, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

Urbana, Illinois

## ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my advisor, Professor T. N. Trick, for his continuing guidance and encouragement through the course of this research. I would also like to thank Professor I. N. Hajj, for his encouragement and helpful suggestions, and Dave Smart, for his valuable discussions on the SLATE program. I wish to express my gratitude to the Digital Equipment Corporation and to the Joint Services Electronics Program for their financial support during my graduate studies.

Finally, I wish to thank my mother for her love and support throughout the course of this work.

# TABLE OF CONTENTS

# CHAPTER 1

# INTRODUCTION

Circuit simulation has become an indispensable tool in the design of integrated circuits. Standard circuit simulators, such as SPICE [1], can predict accurately the circuit performance. However, the use of these simulators is limited to circuits of several hundred transistors. As the size and complexity of the integrated circuit increase, the memory and cpu time requirements for such an analysis become prohibitive.

An effective way to improve the speed and reduce the memory requirement of the circuit simulator is to exploit the modular and repetitive nature of the digital circuits. Digital circuits are mostly designed in a hierarchical fashion with the same basic cells (e.g. logic gates) repeated many times to form the entire circuit. In a large circuit often only a small percentage of the cells will be actively changing states at the same time, while the other cells will remain inactive or "latent".

The circuit simulation program SLATE [3] (a Simulator with Latency and Tearing) developed by Ping Yang at the University of Illinois takes advantage of the properties of the circuits mentioned above to enhance its performance. SLATE utilizes node tearing [5], [6], to partition the circuit into blocks of subcircuits that can be analyzed independently. In the analysis, latent subcircuits are by-passed, which results in a significant savings of the analysis time.

Despite the hierarchical nature of the VLSI circuits, the original version of SLATE allowed only one level of subcircuits, and the nesting of subcircuits to form a larger subcircuit is not allowed.

In this research, three schemes have been proposed to modify SLATE so that it can analyze circuits with subcircuit nesting. The effects of the schemes on the performance of SLATE will be evaluated.

Chapter 2 reviews the problems of circuit simulation and gives a comparison between the "standard" simulators and SLATE. Chapter 3 discusses the node tearing method and the latency scheme used in SLATE. Chapter 4 discusses the three schemes to process nested subcircuits and presents some experimental results on how the performance of SLATE is affected at different degrees of network tearing and latency exploitation. Chapter 5 presents the conclusions. Finally, the Appendix contains the program reference guide for SLATE.

# CHAPTER 2

# REVIEW OF CIRCUIT SIMULATION

With the exception of highly constrained design methodology, it is usually impossible for a circuit designer to produce a guaranteed error-free design with known performance under a wide range of operating conditions. However, with the aid of the ''standard'' circuit simulators such as SPICE [1], designers can predict the voltage and current waveforms of a large variety of circuits accurately and optimize their designs.

However, the speed of the simulator is traded-off with its versatility and accuracy. In order to simulate a wide range of circuits accurately and be adaptable to new technologies that will arise in the future, the program has to use a general algorithm of solving a system of coupled, nonlinear, ordinary differential equations to derive the solution of the circuit equations. Hence, it cannot exploit the special characteristics of a technology to enhance its performance.

In this chapter we shall review some basic techniques that are used in ''standard'' circuit simulation and compare them to the algorithms used in SLATE.

## 2.1. Techniques of Standard Circuit Simulation

The behavior of the circuit being analyzed is described by a set of differential equations:

$$f(\dot{x}(t), x(t), u) = 0 \qquad (2.1)$$
$$x(0) = x_0$$

where x is the unknown variable vector at time t with the initial condition $x_0$ at $t=0$, while u is the input vector and f is a continuous function.

The "standard" circuit simulators are characterized by their use of the following algorithms in the process of solving equation (2.1):

(1) The derivative $\dot{x}(t)$ is replaced by a stiffly stable implicit integration formula, which is a function of $x(t)$.

(2) The time step h and integration order K are controlled automatically to insure the accuracy of the solution.

(3) A quadratically convergent Newton's method is used to solve the resulting system of nonlinear equations.

(4) The system of linear equations involved in each Newton step is solved by sparse Gaussian elimination.

After applying the implicit integration formula, equation (2.1) is transformed into a sequence of nonlinear algebraic equations at various time points in the form of

$$g(x_i) = 0 \qquad (2.2)$$

where the $x_i$'s are the unknown voltage and current values at the time point $t_i$ in the time interval we want to analyze.

Starting at t=0, the nonlinear algebraic equations of (2.2) are linearized by applying the Newton-Raphson method to a set of matrix equations:

$$Ax = b \tag{2.3}$$

Equation (2.3) is solved by Gaussian elimination. Iteration is carried out until the solution has converged or the iteration count limit is exceeded. The program then uses the solution at the present time to predict the solution at the next time point in order to initialize the iteration process at that time point. The process repeats itself until the solution at the last time point is found.

The matrix A is set up using the Modified Nodal Approach (MNA) [8]. Sparse Tableau techniques are applied to reduce the number of operations needed by the Gaussian elimination to solve for x in (2.3).

## 2.3. Problems of Standard Circuit Simulation

Although the "standard" circuit simulator has been successful in the past, it still faces many problems, such as:

(1) As the size of the circuit increases, the cpu time and memory requirements become prohibitive.

(2) The solution of the circuit equations will fail if a zero pivot is chosen in the LU factorization process.

(3) Analysis time is wasted on computing the solutions of parts of the circuit that are not actively changing states.

(4) The form of solution procedure used is not suitable for implementation in machines with parallel processing capabilities.

In the following section we shall discuss the approaches used in the program SLATE as an effort to alleviate the above problems.

## 2.4. Algorithms used in SLATE

The circuit simulation program SLATE was originally developed by Ping Yang at the University of Illinois and then modified later by the same author at the Central Research Laboratory of Texas Instruments. The program contains various features that alleviate the problems of the "standard" circuit simulator.

Firstly, the program uses a reordering scheme [3] that avoids the possibilities of creating zero pivots in the LU factorization process. With a little extra time spent on the preprocessing phase, the reliability and accuracy of the equation solution are substantially improved.

Secondly, the latency of the circuit is exploited. Using node tearing [5], the circuit to be analyzed is partitioned into individual subcircuits and the "rest of the circuit". Each subcircuit can be analyzed independently and the analysis of the latent subcircuits (including the interconnections between the subcircuit and the rest of the circuit) can be by-passed. It was shown that latency exploitation can result in savings of up to 50% of the analysis time [3],[4].

Thirdly, the repetitive nature of the subcircuit definitions is exploited. Each of the subcircuit definitions, which may appear many times, is reordered once and their matrix pointers are only generated once.

# CHAPTER 3

# TEARING DECOMPOSITION

The idea of tearing decomposition is to "tear" the circuit to be analyzed into smaller subcircuits that can be analyzed independently and combine the solutions of those subcircuits together to form the solution of the entire circuit. There are two types of tearing techniques: (1) branch tearing that selects a set of tearing branches and uses their currents as the tearing variables (Fig. 3.1) and (2) node tearing that selects a set of tearing nodes and uses their voltages as the tearing variables (Fig 3.2).

In SLATE, the node tearing method is chosen in favor of branch tearing [3]. It is assumed that the subcircuits will be defined by the user and the parts of the circuit that are not included in a subcircuit definition are automatically assumed to be in the "rest of the circuit" block shown in Fig. 3.1 and Fig. 3.2.

Algebraically, node tearing is equivalent to a special reordering of the circuit equations into a Bordered Block Diagonal Form (BBDF). Each block corresponds to a subcircuit and the border corresponds to the interconnections of the subcircuits.

The BBDF form of matrix reordering has several advantages:

(1) This approach is suitable for the exploitation of latency and parallel processing.

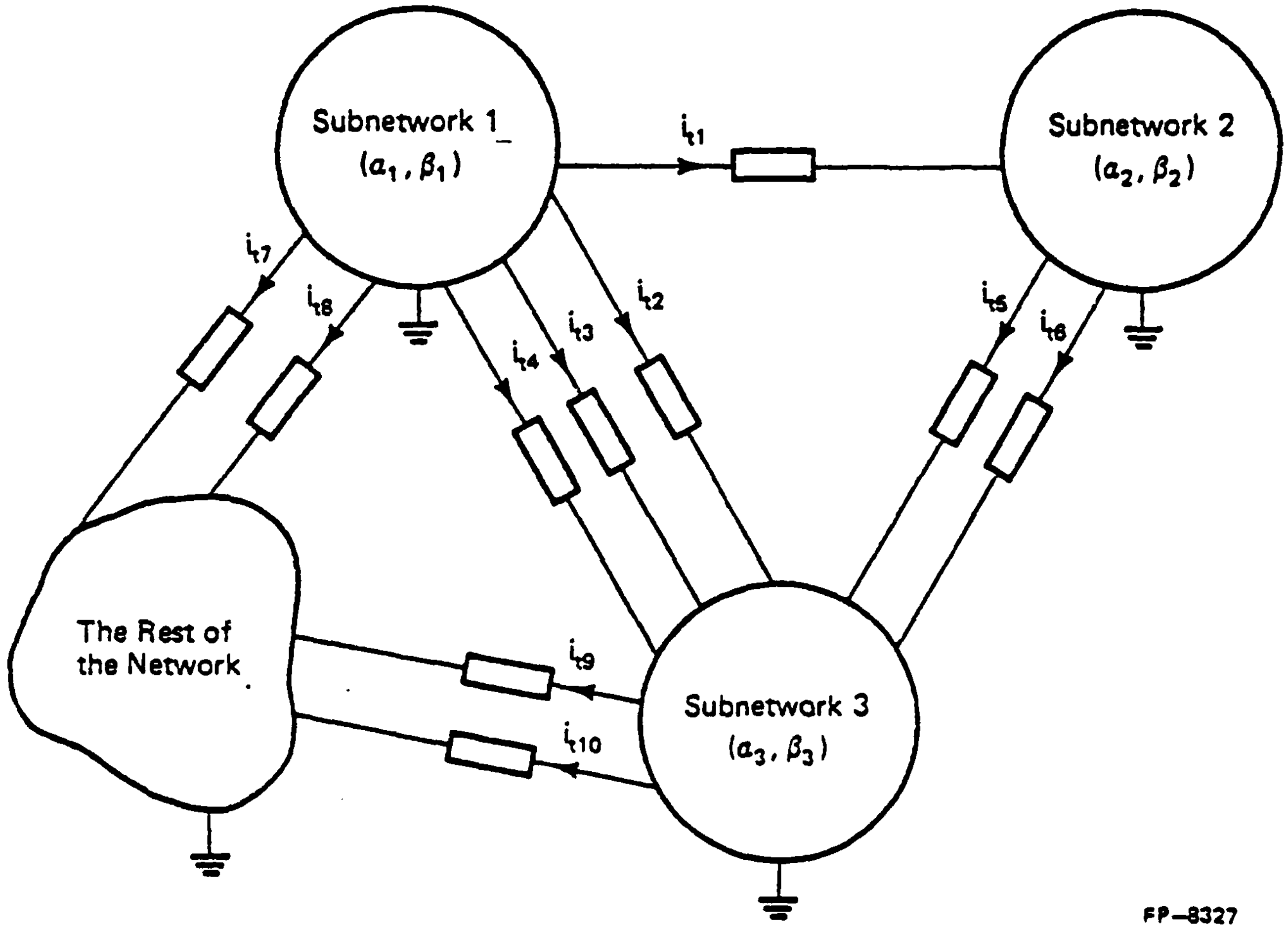


Fig. 3.1 Example of a Network Partitioned into Three Subnetworks by the Branch Tearing Method.

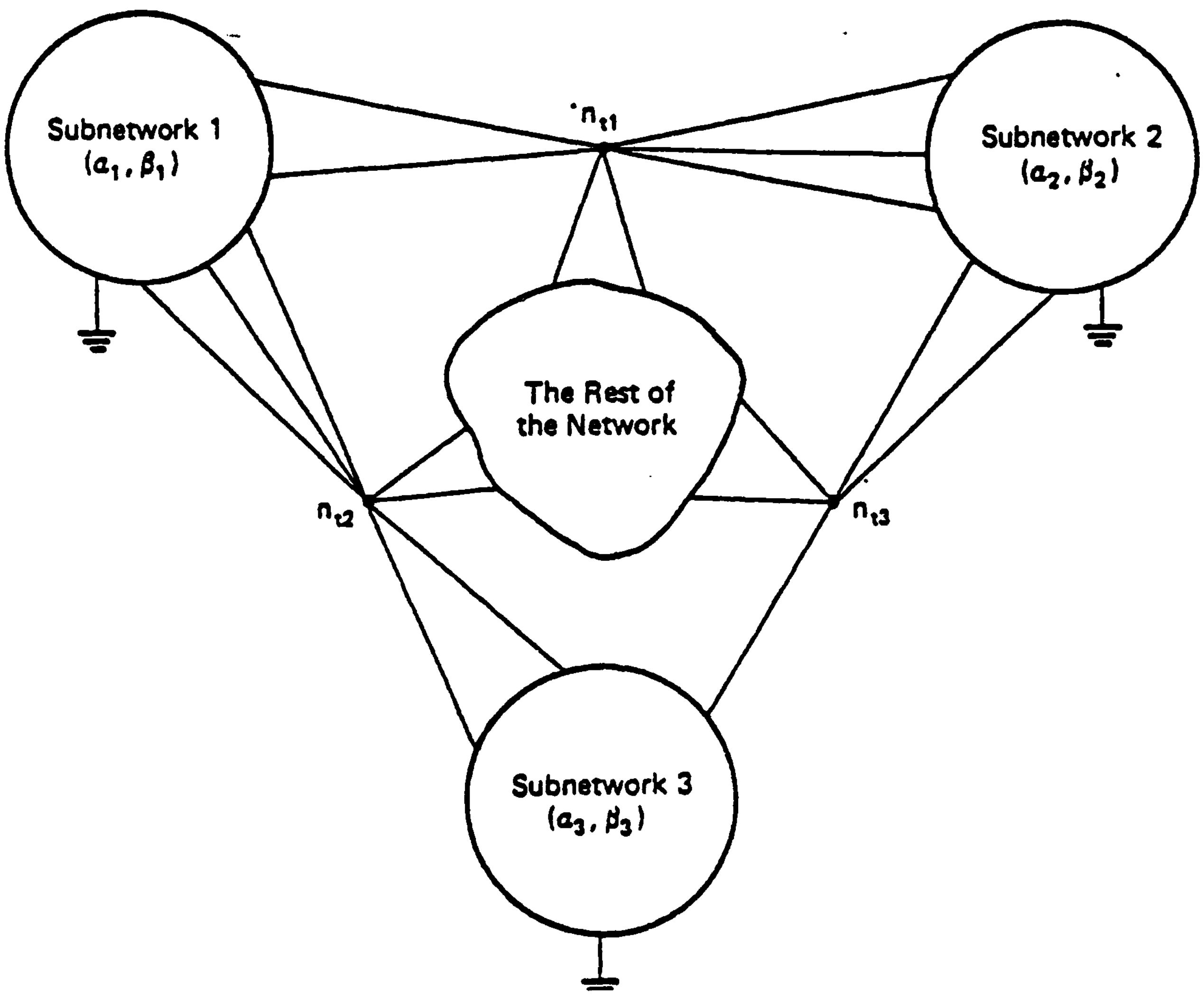


Fig. 3.2 Example of a Network Partitioned into Three Subnetworks by the Node Tearing Method.

(2) The memory requirement is reduced. If storage is limited, individual subcircuits can be loaded into memory and analyzed one by one. This permits a much larger system to be simulated.

In the following sections, we shall discuss the solution procedures of the matrix equations and the latency schemes used in SLATE.

## 3.1. Constructing the Node Tearing Matrix

Consider the network N shown in Fig. 3.2. It consists of k subnetworks { $N_1$, $N_2$, ..., $N_k$}, with subcircuit node sets {$\alpha_1$, $\alpha_2$,..., $\alpha_k$} and sets of tearing nodes {$\alpha_{t1}$, $\alpha_{t2}$, ..., $\alpha_{tk}$}. These subnetworks are connected together with the rest of the circuit which have node set $\alpha_r$. Assuming there is no coupling between the subnetworks, the nodal equations of N can be expressed as:

$$\left[\begin{array}{cccc|c|c}
\underset{\sim}{Y}_{s1} & & & \underset{\sim}{0} & \underset{\sim}{Y}_{st1} & \\
 & \underset{\sim}{Y}_{s2} & & & \underset{\sim}{Y}_{st2} & \underset{\sim}{0} \\
 & \underset{\sim}{0} & \ddots & & & \\
 & & & \underset{\sim}{Y}_{sk} & \underset{\sim}{Y}_{stk} & \\
\hline
\underset{\sim}{Y}_{ts1} & \underset{\sim}{Y}_{ts2} & & \underset{\sim}{Y}_{tsk} & \underset{\sim}{Y}_{tt} & \underset{\sim}{Y}_{tr} \\
\hline
 & \underset{\sim}{0} & & & \underset{\sim}{Y}_{rt} & \underset{\sim}{Y}_{rr}
\end{array}\right]
\left[\begin{array}{c}
\underset{\sim}{v}_{s1} \\ \underset{\sim}{v}_{s2} \\ \vdots \\ \underset{\sim}{v}_{sk} \\ \hline \underset{\sim}{v}_{t} \\ \hline \underset{\sim}{v}_{r}
\end{array}\right]
=
\left[\begin{array}{c}
\underset{\sim}{J}_{ss1} \\ \underset{\sim}{J}_{ss2} \\ \vdots \\ \underset{\sim}{J}_{ssk} \\ \hline \underset{\sim}{J}_{ts} \\ \hline \underset{\sim}{J}_{rs}
\end{array}\right] \qquad (3.1)$$

The matrix equations of (3.1) are solved by LU factorization and forward backward substitutions. There are several possible factorization and substitution schemes [6]. The LU factorization scheme $S_1$ and substitution scheme $F_1$ are used (Tables 3.1, 3.2).

Equation (3.1) is solved first by eliminating all the $Y_{tsi}$ (step 1 of $S_1$) to get the interconnection matrix equations:

$$\begin{bmatrix} \underset{\sim}{Y}^{*}_{tt} & \underset{\sim}{Y}_{ts} \\ \underset{\sim}{Y}_{rt} & \underset{\sim}{Y}_{rr} \end{bmatrix} \begin{bmatrix} \underset{\sim}{v} \\ \underset{\sim}{v}_{r} \end{bmatrix} = \begin{bmatrix} \underset{\sim}{J}^{*}_{ts} \\ \underset{\sim}{J}_{rs} \end{bmatrix} \qquad (3.2)$$

where $Y^{*}_{tt} = Y_{tt} - \sum_{i=1}^{k} Y_{tsi}(Y_{si})^{-1}Y_{tsi}$

and $J^{*}_{ts} = J_{ts} - \sum_{i=1}^{k} Y_{tsi}(Y_{si})^{-1}J_{ssi}$

Equation (3.2) can be solved to obtain $v_t$ and $v_r$ (step 3 of $S_1$) and the solution of the $v_{si}$ can be obtained by backward substitution (steps 4, 5 and 6 of $S_1$).

## 3.2. Latency Scheme of SLATE

There are two types of latency, namely, latency in the Newton Raphson iteration and latency in time, their nature and origins are

Table 3.1 Factorization Scheme $F_1$ used in SLATE

$$\begin{bmatrix} \underset{\sim}{L}_{sk} & \underset{\sim}{0} \\ \underset{\sim}{W}^T & \underset{\sim}{L}_{tk} \end{bmatrix} \begin{bmatrix} \underset{\sim}{U}_{sk} & \underset{\sim}{V} \\ \underset{\sim}{0} & \underset{\sim}{U}_{tk} \end{bmatrix}$$

$$\underset{\sim}{L}_{sk} \underset{\sim}{U}_{sk} = \underset{\sim}{Y}_{sk} \qquad \underset{\sim}{V} = \underset{\sim}{L}_{sk}^{-1} \underset{\sim}{Y}_{stk}$$

$$\underset{\sim}{W}^T = \underset{\sim}{Y}_{tsk} \underset{\sim}{U}_{sk}^{-1}$$

and

$$\underset{\sim}{L}_{tk} \underset{\sim}{U}_{tk} = \underset{\sim}{Y}_{ttk}^{*} - \underset{\sim}{Y}_{tsk} \underset{\sim}{U}_{sk}^{-1} \underset{\sim}{L}_{sk}^{-1} \underset{\sim}{Y}_{tsk}$$

Table 3.2 Substitution Scheme $S_1$ used in SLATE

| Step | $S_1$ |
|---|---|
| 1 | $\underset{\sim}{L}_{sk} \underset{\sim}{a} = \underset{\sim}{J}_{ssk}$ |
| 2 | $\underset{\sim}{y} = \underset{\sim}{W}^T \underset{\sim}{a}$ |
| 3 | $\underset{\sim}{L}_{tk} \underset{\sim}{U}_{tk} \underset{\sim}{v}_{tk} = \underset{\sim}{J}_{tsk} - \underset{\sim}{y}$ |
| 4 | $\underset{\sim}{Z}_1 = \underset{\sim}{V} \underset{\sim}{v}_{tk}$ |
| 5 | $\underset{\sim}{\hat{a}} = \underset{\sim}{a}_1 - \underset{\sim}{Z}_1$ |
| 6 | $\underset{\sim}{U}_{sk} \underset{\sim}{v}_{sk} = \underset{\sim}{\hat{a}}$ |

explained in detail in pp. 138-141 of [3]. For latency in the Newton Raphson iteration, subcircuit $N_k$ is latent at the i'th iteration point if:

$$(1)\ |V_{Nkm}(i-1)-V_{Nkm}(i-2)| \le \varepsilon_a+\varepsilon_r \max(|V_{Nkm}(i-1)|, |V_{Nkm}(i-2)|) \quad \text{for } m=1,2,\ldots \qquad (3.3)$$

and

$$(2)\ |V_{tkm}(i)-V_{tkm}(i-1)| \le \varepsilon_a+\varepsilon_r \max(|V_{tkm}(i)|, |V_{tkm}(i-1)|) \quad \text{for } m=1,2,\ldots \qquad (3.4)$$

where $\varepsilon_a$ and $\varepsilon_r$ are the absolute and relative tolerance, respectively, and $V_{Nkm}(i)$ are the subcircuit and tearing node voltages at the i'th iteration. The subcircuit $N_k$ will remains latent as long as all its external nodes remain latent:

$$(3)\ |V_{tkm}(i+j)-V_{tkm}(i-1)| \le \varepsilon_a+\varepsilon_r \max(|V_{tkm}(i+j)|, |V_{tkm}(i-1)|) \quad \text{for } m=1,2,\ldots;\ j=1,2,\ldots \qquad (3.5)$$

The scheme 2 proposed in [3] is used to check for latency in time. A subcircuit $N_k$ is declared to be latent in time if its tearing node voltages at times $t_n$ and $t_{n-1}$ satisfy:

$$(1)\ |V_{tkm}(t_n)-V_{tkm}(t_{n-1})|\leq\varepsilon_a+\varepsilon_r\max(|V_{tkm}(t_n)|,\ |V_{tkm}(t_{n-1})|) \qquad \text{for } m=1,2,\ldots \qquad (3.6)$$

and the currents of the energy storage elements:

$$(2)\ |I_{km}(t_n)-I_{km}(t_{n-1})|\leq\varepsilon_c+\varepsilon_r\max(|I_{km}(t_n)|,\ |I_{km}(t_{n-1})|) \qquad m=1,2\ldots b \qquad (3.7)$$

where $\varepsilon_c$ is the absolute error tolerance of the current. This condition is used to check if the changes of the energy storage elements are small. Furthermore,

$$(3)\ h_{n-1}|I_{km}(t_n)-I_{km}(t_{n-1})|/|Q_{km}(t_n)-Q_{km}(t_{n-1})|\geq 1 \qquad m=1,2\ \ldots \qquad (3.8)$$

where $I_{km}(t_n)$ is the current (or voltage) and $Q_{km}(t_n)$ is the charge (or flux) of the capacitor or inductor checked at time $t_n$ and $h_{n-1}$ is the time step used. This condition is used to check if there are slowly varying nodes within $N_k$.

The subcircuit $N_k$ will remain latent as long as:

$$(4)\ |V_{tkm}(t_{n+j})-V_{tkm}(t_{n-1})|\leq\varepsilon_a+\varepsilon_r\max(|V_{tkm}(t_{n+j})|,\ |V_{tkm}(t_{n-1})|) \qquad \begin{matrix} m=1,2,\ldots \\ j=1,2,\ldots \end{matrix} \qquad (3.9)$$

# CHAPTER 4

# PROCESSING NESTED SUBCIRCUITS AND TEARING CONSIDERATIONS

When there is more than one level of subcircuit (Fig. 4.1) nesting in the circuit description, the problems of setting up the matrix structure and solving the matrix equations become more complicated. In our research three schemes of processing the nested subcircuit structures have been considered:

(1) Implement a nested BBDF matrix that can exploit latency in a hierarchical manner.

(2) Tear the nested subcircuits away from their parents to form a circuit structure with only one level of subcircuit.

(3) Expand the nested subcircuits inside the first level subcircuit definition to form a circuit structure with only one level of subcircuits.

## 4.1. Scheme 1: The Nested BBDF Matrix

The nodal equations of the circuit to be analyzed can be reordered into nested BBDF form. The interconnections of the first level of subcircuit are placed on the outermost border of the matrix and the interconnections of the second level of nested subcircuits are placed on the next level of border and so forth. The reordering repeats up to the last level of nesting.

For example, the circuit in Fig 4.1 has a maximum of three levels of nesting. Its BBDF equation matrix is shown in Fig 4.2.

If this form of matrix equation is adopted, latency of the subcircuits can be exploited in a hierarchical manner during the analysis of the circuit. The program checks the latency of the subcircuits at different levels of the circuit hierarchy and by-passes the analysis of the subcircuits starting at the level that they were found to be latent.

However, this scheme for processing the nested subcircuits has several disadvantages. Firstly, it is difficult to implement. The equation matrix and solution procedure become complex with arbitrary levels of nesting. Secondly, the cpu time saved in hierarchical latency exploitation may not compensate for the overhead introduced by the extra latency checking required.

## 4.2. Scheme 2: Tearing the Nested Subcircuits

In the second scheme, the subcircuits nested inside other subcircuits are torn away from their parents to form modified subcircuits with their tearing nodes placed on the border of the equation matrix. The tearing procedure continues until all nested subcircuits are levelized.

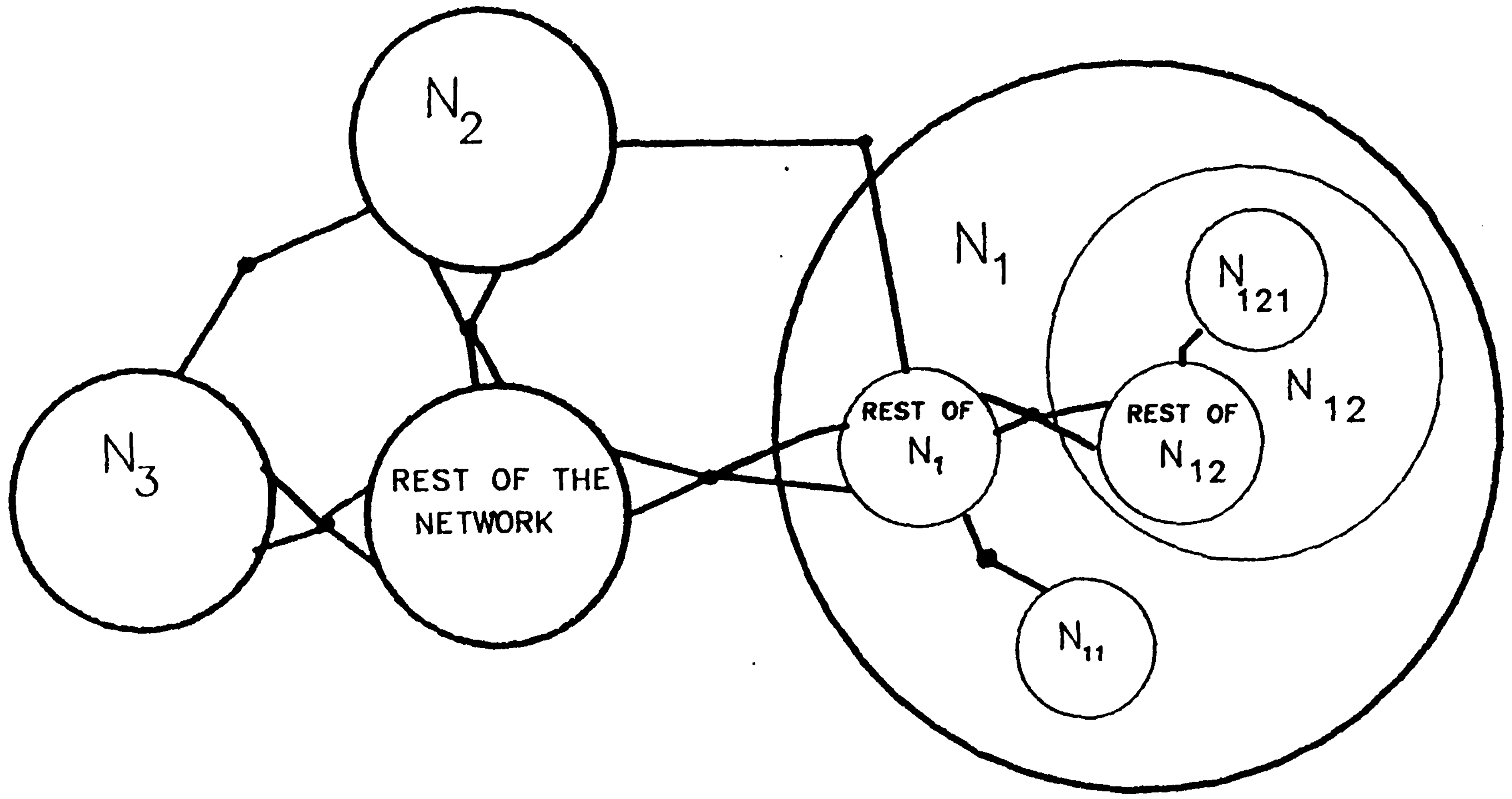


Fig.4.1 An Example of a Network with Nested Subnetworks.

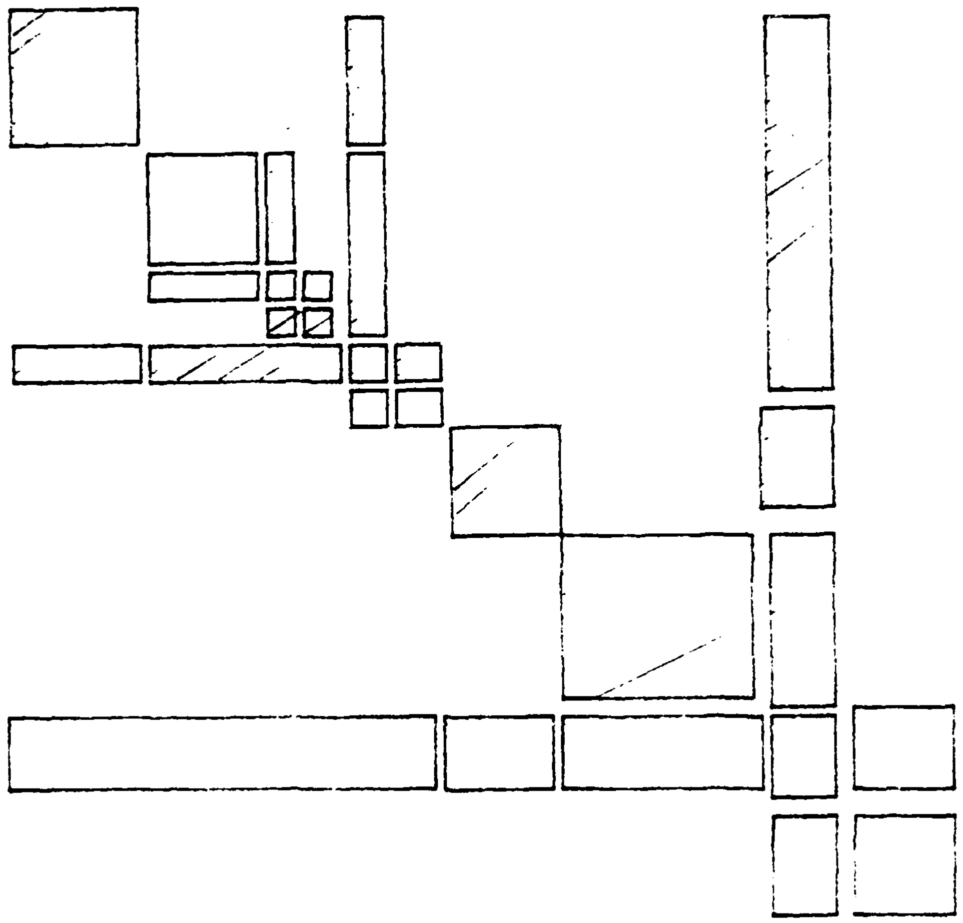


Fig. 4.2 The Nested BBDF Matrix of the Network shown in Fig. 4.1.

For example, the subcircuit $N_1$ in Fig. 4.1 contains the second level nested subcircuits $N_{11}$, $N_{12}$. $N_{12}$ contains the third level subcircuits $N_{121}$. These subcircuits are torn apart to form modified subcircuits without any nesting. Algebraically, this is equivalent to reordering the tearing nodes of the nested subcircuits on the border of the nodal equation matrix:

$$\left[\begin{array}{cccccc|c|c}
\underset{\sim}{Y}'_{s1} & & & & & & \underset{\sim}{Y}'_{st1} & \\
& \underset{\sim}{Y}_{s2} & & \underset{\sim}{0} & & & \underset{\sim}{Y}_{st2} & \underset{\sim}{0} \\
& & \underset{\sim}{Y}_{s3} & & & & \underset{\sim}{Y}_{st3} & \\
& & & \underset{\sim}{Y}_{s11} & & & \underset{\sim}{Y}_{st11} & \\
& \underset{\sim}{0} & & & \underset{\sim}{Y}'_{s12} & & \underset{\sim}{Y}'_{st12} & \\
& & & & & \underset{\sim}{Y}_{s121} & \underset{\sim}{Y}_{st121} & \\
\hline
\underset{\sim}{Y}'_{ts1} & \underset{\sim}{Y}_{ts2} & \underset{\sim}{Y}_{ts3} & \underset{\sim}{Y}_{ts11} & \underset{\sim}{Y}_{ts12} & \underset{\sim}{Y}_{ts121} & \underset{\sim}{Y}'_{tt} & \underset{\sim}{Y}_{tr} \\
\hline
& \underset{\sim}{0} & & & & & \underset{\sim}{Y}_{rt} & \underset{\sim}{Y}_{rr}
\end{array}\right]
\left[\begin{array}{c}
\underset{\sim}{v}'_{s1} \\ \underset{\sim}{v}_{s2} \\ \underset{\sim}{v}_{s3} \\ \underset{\sim}{v}_{s11} \\ \underset{\sim}{v}'_{s12} \\ \underset{\sim}{v}_{s121} \\ \hline \underset{\sim}{v}_{t} \\ \hline \underset{\sim}{v}_{r}
\end{array}\right]
=
\left[\begin{array}{c}
\underset{\sim}{J}'_{ss1} \\ \underset{\sim}{J}_{ss2} \\ \underset{\sim}{J}_{ss3} \\ \underset{\sim}{J}_{ss11} \\ \underset{\sim}{J}'_{ss12} \\ \underset{\sim}{J}_{ss121} \\ \hline \underset{\sim}{J}_{ts} \\ \hline \underset{\sim}{J}_{rs}
\end{array}\right] \tag{4.1}$$

After tearing, the nested subcircuits $N_{11}$ and $N_{12}$ are torn away from $N_1$; $N_{121}$ is torn away from $N_{12}$ and the tearing nodes of these nested subcircuits are placed on the border of the matrix. This scheme has the following characteristics:

(1) It is easy to implement. The subcircuit tearing can be done in the preprocessing phase and the analysis part of SLATE need not

be changed.

(2) The number of subcircuits and tearing nodes is usually large and the sizes of the subcircuits are small; the maximum amount of tearing is carried out.

(3) More latency checking is required than for scheme 3.

(4) The percentage of latency exploitation is high. Since the subcircuits are smaller and more numerous, it is more likely to find a latent subcircuit.

## 4.3. Scheme 3: Levelizing the Subcircuits

In this scheme all the nested subcircuits inside other subcircuits are expanded to form a circuit with only one level of subcircuits with new elements from the nested subcircuit calls added to the "parent" subcircuit. It has several characteristics.

(1) It is easy to implement; subcircuit expansion can be done in the preprocessing phase.

(2) There is less tearing since the nested subcircuits are expanded instead of torn apart as in scheme 2.

(3) Less latency checking is required than for scheme 2.

(4) The percentage of latency exploitation is usually less than scheme 2. Since the subcircuits are larger and less numerous, it is less likely to find a latent subcircuit.

## 4.4. Experimental Results

To choose between schemes 2 and 3 we shall consider the tradeoff between the amount of tearing and the savings in analysis time. The amount of cpu time the program spent in transient analysis can be divided into three classes:

(1) The time used in evaluating the device models. (This is not affected by node tearing.)

(2) The time used in solving the matrix equations using LU factorization and forward and backward substitutions.

(3) The time spent on checking the latency of the subcircuits.

In most cases, the time spent on (3) is negligible compared to (2). However, if the circuit is large and has a lot of latency, scheme 3 will be more efficient than scheme 2. Since the subcircuits generated by scheme 3 is larger, less redundant latency checking is needed. On the contrary, if the circuit is more active, scheme 2 will be more efficient than scheme 3 since it can exploit latency more effectively using smaller subcircuits.

The following examples will show that the program user should partition the circuit appropriately in order to maximize the benefits that can be gained from latency exploitation.

Example 4.1

In this example the full adder circuit is analyzed. The adder cell is implemented using CMOS logic gates (Fig. 4.3). There are three levels of tearing:

(1) Level 1: The full adder cell is not partitioned, 44 MOS transistors are connected together to form a cell with 6 tearing nodes.

(2) Level 2: The cell is partitioned into 2 XOR and 3 NAND gates with 9 tearing nodes.

(3) Level 3: The cell is partitioned into 9 NAND gates and 4 inverters with 17 tearing nodes.

Note that this is not the most efficient way to form the full adder circuit, this implementation is used so that tearing and subcircuit nesting can be illustrated more clearly. The XOR, NAND and inverters consist of 16, 4 and 2 MOS transistors respectively.

Two types of input waveforms were applied to test the effects of tearing on the analysis time:

(1) All the input bits to the adder cell rise from 0 volts to 5 volts. This will cause the maximum amount of circuit activity.

(2) Only one input bit to the adder cell changes from 0 volts to 5 volts. The circuit should have a larger amount of latency.

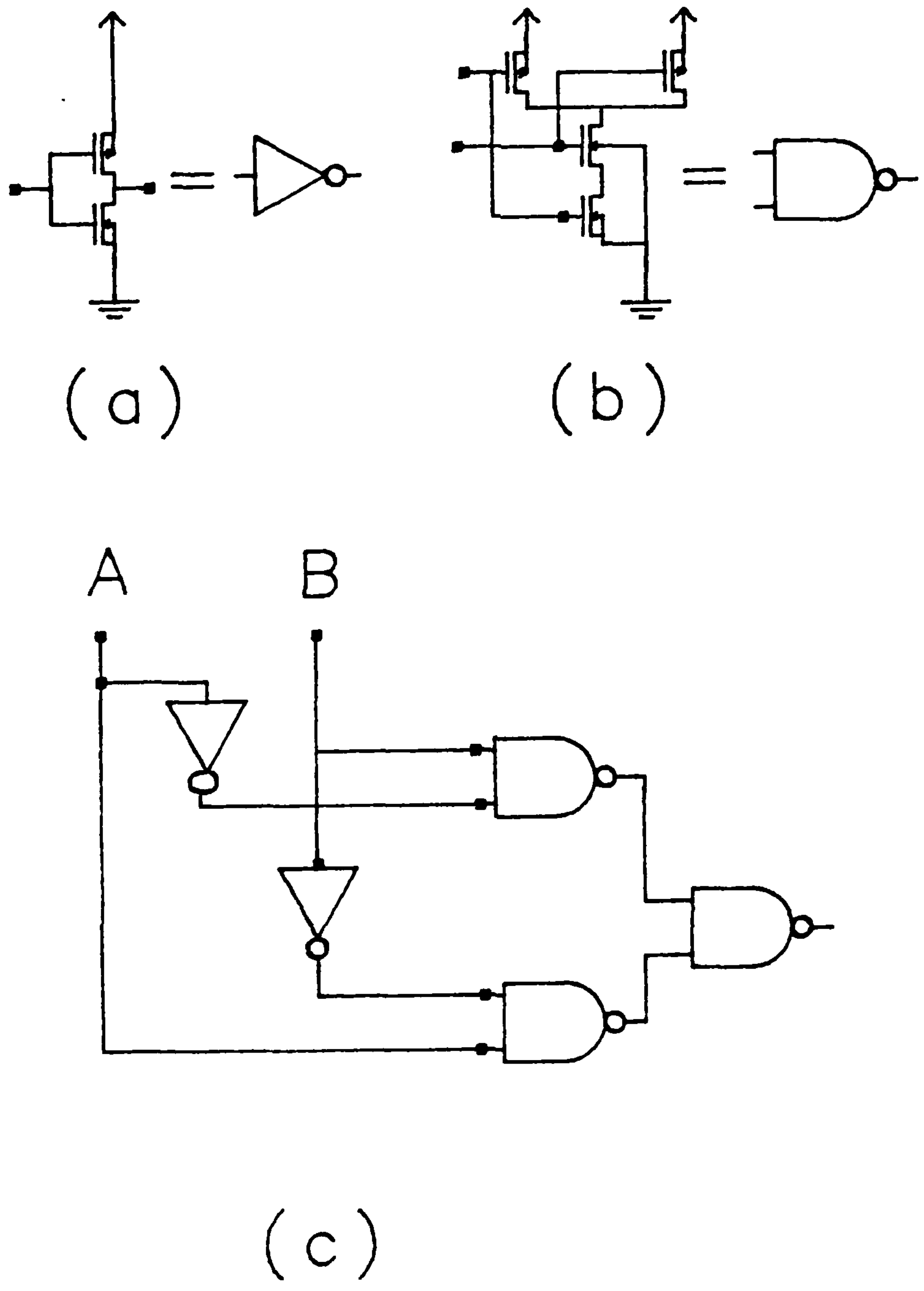
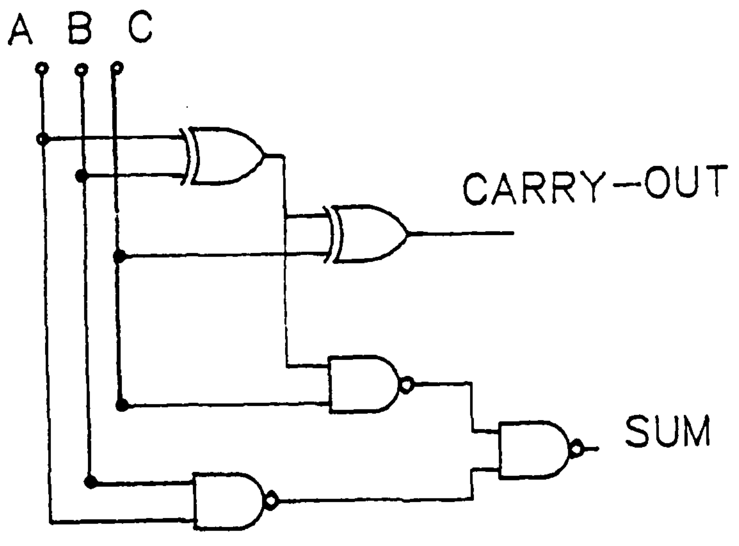
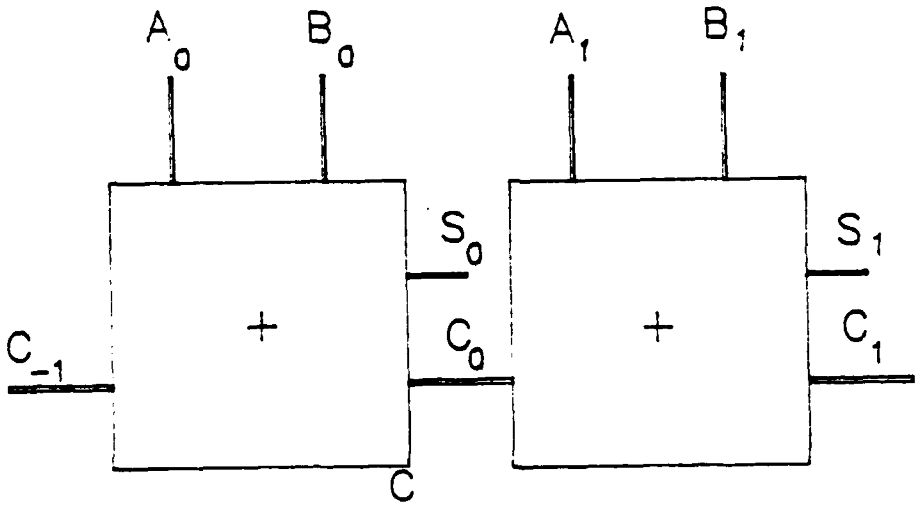


Fig.4.3 (a) CMOS Inverter (b) CMOS Two Input NAND Gate
(c) XOR Gate



(d)



(e)

Fig.4.3 (cont'd.) (d) One Bit Full Adder Cell (e) Two Bit Full Adder Circuit.

One and two bit full adders were simulated, the results are summarized below in Tables 4.1, 4.2, 4.3 and 4.4:

Table 4.1 One Bit Adder Circuit - All Inputs Changing
(Analysis time = 20 ns, 44 MOS transistors)

| Tearing Level | INLATN | ITOTAL | NUMNIT | TRANAN | % LATENCY |
|---|---|---|---|---|---|
| 1 | 76 | 76 | 76 | 38.22s | 0. |
| 2 | 297 | 380 | 76 | 35.15s | 21.76 |
| 3 | 683 | 988 | 76 | 34.57s | 30.84 |

Table 4.2 One Bit Adder Circuit - One Input Changing
(Analysis time = 20 ns, 44 MOS transistors)

| Tearing Level | INLATN | ITOTAL | NUMNIT | TRANAN | % LATENCY |
|---|---|---|---|---|---|
| 1 | 77 | 77 | 77 | 38.35s | 0. |
| 2 | 267 | 390 | 78 | 32.08s | 31.54 |
| 3 | 596 | 988 | 76 | 28.70s | 39.67 |

Table 4.3 Two Bit Adder Circuit - All Inputs Changing
(Analysis time = 40 ns, 88 MOS transistors)

| Tearing Level | INLATN | ITOTAL | NUMNIT | TRANAN | % LATENCY |
|---|---|---|---|---|---|
| 1 | 114 | 122 | 122 | 115.73 | 6.08 |
| 2 | 486 | 700 | 140 | 116.85 | 30.64 |
| 3 | 1039 | 1495 | 115 | 98.63 | 30.52 |

Table 4.4 Two Bit Adder Circuit - One Input Changing
(Analysis time = 40 ns, 88 MOS transistors)

| Tearing Level | INLATN | ITOTAL | NUMNIT | TRANAN | % LATENCY |
|---|---|---|---|---|---|
| 1 | 120 | 126 | 126 | 121.57 | 5.00 |
| 2 | 468 | 630 | 126 | 108.05 | 25.69 |
| 3 | 1256 | 1976 | 152 | 117.12 | 36.42 |

where

INLATN = Number of nonlatent subcircuits times the number of iterations

ITOTAL = Number of subcircuits times the number of iterations

NUMNIT = Number of iterations in the transient analysis

TRANAN = cpu time spent on transient analysis (seconds)

% LATENCY = 100% times ( 1 - INLATN/ITOTAL ) is the

measure of latency exploitation in
the program.

We note that when the input is very active (all inputs changing) and the circuit is small (one bit adder), it is more advantageous to partition the circuit into smaller blocks so that latency can be exploited effectively.

However, if the circuit is larger and less active, the overhead introduced by latency checking can be significant. In the two bit adder with one input changing circuits, the circuit with lots of tearing (level 3) has many tearing nodes and a large equation matrix border. Although the percentage of latency exploitation is much higher than the circuit with less tearing (level 1), there is almost no savings in analysis time.

In the worst case, the overhead spent on latency checking can outweigh the savings in cpu time gained through by-passing the analysis of the latent subcircuits. Thus, it is important for the program user to tear the circuit in the appropriate sizes and at the appropriate nodes in order to maximize the benefits that can be gained from SLATE.

# CHAPTER 5

# CONCLUSIONS

In this report we reviewed the algorithms and some of the common problems that are faced by the ''standard'' simulators. We then introduced the SLATE program that uses latency and tearing techniques to alleviate those problems. → to 1473

The problem of nested subcircuits was investigated. Three schemes were proposed to process the nested subcircuit structures. The amount of tearing and latency checking differs in each scheme. Some sample circuits have been tested to study the effects of the amount of tearing on the performance of the simulator. The results showed that the program user should partition the circuit appropriately in order to gain the maximum savings of analysis time in the SLATE program.

# APPENDIX

## PROGRAM REFERENCE GUIDE FOR SLATE

The purpose of this appendix is to describe the organization, data structure and table specifications of SLATE. Program users should refer to the appendix of [4] for a full description of the functions and usage of the program.

### A.1. Introduction

SLATE is a general purpose circuit simulation program that performs nonlinear dc and nonlinear transient circuit analyses. This appendix describes the internal design of the program. For the fundamental theory and device models used, refer to [3] and [4].

The SLATE program consists of 18,000 Fortran77, C and assembler statements divided into six major overlays: READIN, ERRCHK, SETUP, DCTRAN, DCOP and OVTPVT. Since the program was developed from SPICE2 their program organization and data structures are very similar. Hence, this report will focus on describing the parts of SLATE that are different from SPICE2 (i.e., the SETUP and DCTRAN overlays that utilize node tearing and latency exploitation). For a full description of the rest of the program and the tables and common blocks used, refer to the SPICE2 Report [7].

Section A.2. briefly describes the dynamic data management techniques that are used in the program. It can be omitted by readers that are already familiar with the memory management techniques used in SPICE2. Section A.3. describes the overlay root. Sections A.4. and A.5. describe the readin and error checking procedures; Sections A.6. and A.7. describe the setup and analysis procedures in detail. Sections A.8. and A.9. gives a summary of the table and common blocks that are used in SLATE. Finally, a sample circuit description using nested subcircuit calls is included. To conserve space, the tables and common locks that are also present in SPICE2 are not listed again.

## A.2. Memory Management System

With the exception of most flags, all the data in SLATE are stored in the form of managed tables in the /BLANK/ array VALUE, which can be redimensioned in the main program according to the memory availability at each user site. The VAX release version of SLATE assumes the virtual memory feature and dimensions VALUE() to 100,000 double precision words.

The memory manager in SLATE controls "tables" using "table pointers". A "table" is a block of memory; a "table pointer" is a variable which serves to identify a block and to indicate the origin of the block in memory. In SLATE, all the data values are stored in the array VALUE or NODPLC, which are declared to be equivalent. NODPLC stores integer data while VALUE stores double precision data. As an example, suppose we have a table IUR of size 100 which contains integer data and another table LXY of size 50 which contains double precision real data. Then the contents of these tables can be accessed as nodplc(IUR+1) through nodplc(IUR+100) and value(LXY+1) through value(LXY+50), respectively.

The set of procedures that is used to manipulate these tables are:

| entry | description |
|---|---|
| SETMEM | initialize the memory manager. |
| GETMEM(P,S) | makes a new managed table of (words) size S pointed to by tp (table pointer) P. |
| RELMEM(P,S) | reduces by S words the size of the table pointed to by tp P. |
| EXTMEM(P,S) | extends by S words the size of the table |

| | |
|---|---|
| | pointed to by tp P. |
| SIZMEM(P,S) | sets S to the size in words of the table pointed to by tp P. |
| PTRMEM(P1,P2) | changes the tp for the tables pointed to by tp P1 and P2. |
| CRUNCH | forces a compaction of the dynamically managed memory. |

A "table entry" table is maintained by the memory manager to keep track of all the tables allocated by the program. This table contains a four word entry for each table allocated of the form:

| word | contents |
|---|---|
| 1 | table origin (array subscript of NODPLC) |
| 2 | allocated table size (in words) |
| 3 | requested block size (in words) |
| 4 | address of table pointer |

Whenever a table management routine is called, it will check for the validity of the table pointer and the size of the table. Any internal error in memory management that is trapped will stop the execution of the program and cause the "table entry" table to be printed by the subroutine DMPMEM.

## A.3. The Overlay Root

The overlay root drives the rest of the program. It calls the first level overlays to process the circuit description, performs error checking, sets up the matrix structure, analyzes the circuit and outputs the results. The root consists of the main program and the subroutines SETMEM, COMPRS, MEMPTR, DMPMEM, TMPUPD, OUTNAM, ALFNUM, FIND, DCOP, MOVE, COPY, ZERO, SECOND and LOCF. A flowchart of the main program is:

```
initialize;
while (.not. end-of-input)
{call READIN; call ERRCHK; call SETUP;
    repeat
         { TEMP=next analysis temperature; call TMPUPD
           if (dc transfer curves requested)
              {MODE=1, MODEDC=3;
               call DCTRAN; call OVTPVT;}
           if (dc small signal operation point requested)
              {MODE=1, MODEDC=1;
               call DCTRAN; call DCOP;}
           if (transient analysis requested)
              {MODE=1, MODEDC=2;
               call DCTRAN; call DCOP;
               MODE=2;
               call DCTRAN; call OVTPVT;}
         }
   until (no more analysis temperature)
}
stop
```

The variables MODE and MODEDC control the type of analysis that is performed; they are discussed in Section A.7.

The READIN overlay reads in the circuit description and sets certain flags that indicate what analyses are requested. It builds the linked lists that store the input data. The ERRCHK overlay performs miscellaneous error checking. The routine renumbers the nodes in ascending order (starting with the ground node as 1) and checks the correctness of the device models used. It also expands all the subcircuit calls and checks the topology of the circuit.

The SETUP overlay constructs the matrix pointers that are used to manipulate the matrix coefficients in the DCTRAN overlay. A reordering scheme that avoids zero pivots is implemented. Node tearing and sparse matrix techniques are used to order the matrix into the bordered block diagonal form (BBDF) that is used later in DCTRAN to facilitate the latency exploitation scheme in analysis.

The DCTRAN overlay performs the dc operating point, dc transfer curve or transient analysis as determined by the variables MODE and MODEDC. Latency properties are being exploited at (1) the time level, (2) the subcircuit level, (3) the device level, and (4) the Newton Raphson level to reduce the analysis time. The details of the latency exploitation scheme are given in Section A.7. Finally, the OVTPVT overlay generates the tabular output listings and the plots from the results of the analysis.

The overlay root contains several utility subroutines that are used throughout the rest of the program. A brief description of them is given below (A and B are array names):

| <u>name</u>(<u>arguments</u>) | <u>description</u> |
|---|---|
| MOVE(A,I,B,J,N) | move N characters from B to A, starting at the J'th and I'th character positions respectively; eight character/word is assumed. |
| COPY4(A,B,N) | move N integers from A to B |

| | |
|---|---|
| COPY8(A,B,N) | COPY4(), but for real variables |
| COPY16(A,B,N) | COPY4(), but for complex variables |
| ZERO4(A,N) | ZERO A(1) to A(N) integers |
| ZERO8(A,N) | ZERO4(), but for real variables |
| ZERO16(A,N) | ZERO4(), but for complex variables |
| LOCF(LTAB) | give the address of the variable LTAB |

## A.4. READIN Overlay

The READIN overlay of SLATE is very similar to the one found in SPICE2. It consists of subroutines READIN, GETLIN, CARD, KEYSRC, EXTNAM, RUNCON, OUTDEF and NXTCHR. This overlay processes the input data and builds the linked lists which describe the circuit to be analyzed and sets certain flags in the common blocks to indicate which analyses have been requested.

### A.4.1. Readin

The subroutine READIN processes elements, device models, subcircuit definitions and the force and initial control cards. The routine first calls SETMEM to initialize the memory management system. It then calls subroutine CARD to read in each card. Subroutine FIND is then called to preset storage for each input element. Analysis and other control cards are processed by RUNCON. A flowchart of this subroutine is:

```
call SETMEM;
read title; if (end-of-file found) exit
initialize;
call CARD;  if (end-of-file found) exit
NSBCKT = 0;
repeat
{
      {element description: process in READIN
      .SUBCKT : set NSBCKT=1,process in READIN
      .ENDS   : set NSBCKT=0,process in READIN
      .FORCE  : process in READIN
      .INITIAL: process in READIN
      .END    : exit
      other "." lines, call RUNCON
      }
```

```
    call CARD;
    }   until (end-of-file reached)
     exit
```

The program sets the value of the variable NSBCKT to one if a subcircuit definition card is encountered and resets it when the subcircuit definition ends. Note that only one level of subcircuit definition is allowed and all model and control cards must not be enclosed within a subcircuit definition. The variables in the common block /FORCE/ will be set if a .FORCE or .INITIAL card is read (see Sectior A.9.).

### A.4.2. Forward References

Since SLATE does not require the circuit description read in an ordered fashion, sometimes an element may be referenced before it is actually read in. For example:

```
      :
    .DC V1 0 5 .5
      : (other inputs)
    V1 6 7 DC
      : (other inputs)
    .END
```

The voltage source V1 is referenced by the ''.DC' statement before it is actually read in. FIND will reserve a linked list element for V1 and then store an undefined flag ''.UN' in nodplc(loc+2) of that element. If V1 is defined later, nodplc(loc+2) will be filled by some integer datum that is not equal to ''.UN', thus, resolving the forward reference problem.

## A.4.3. FIND

The FIND subroutine is used to locate a particular list element or to allocate storage if the element cannot be found. It has four parameters:

| parameter | description |
|---|---|
| ANAME | name of the linked list element |
| ID | id number of the list to search |
| LOC | subscript of NODPLC, set by FIND to the location of the element |
| IFORCE | 0=> no restriction<br>1=> element must not already exist<br>2=> element is created by subcircuit expansion |

A flowchart of FIND is given as follows:

```
if (NSBCKT.eq.0) then
        element in the nominal circuit or device model card
        search LOCATE(ID) list for ANAME;
   else
        element inside a subckt definition
        search (ISBCKT+NSBCKT) list for ANAME;
   endif
if (element not found) then
     {  if (IFORCE.eq.2) then
             reserve space according to COMMON /LNODS/;
           else
             reserve space according to COMMON /LNOD/;
           endif
        set nodplc(loc+2)='.UN';(forward reference)
        if (NSBCKT.ne.0) nodplc(loc-1)=ID;
        return;
     }
   else if (IFORCE.eq.0) then
           not an element definition request, return;
   else if (nodplc(loc+2).eq. '.UN') then
          element already referenced but not read in yet;
          return;
   else
          element defined already,((loc+2).ne.'.UN'),
          attempt to redefine element, set NOGO=1;
          return;
   endif
```

FIND is called by the READIN and RUNCON subroutines during readin of the circuit description and also by the SUBCKT subroutine during subcircuit expansion.

When called by READIN, IFORCE is set to 1. The routine first determines the value of NSBCKT; FIND searches the LOCATE(ID) linked list, if NSBCKT is zero (element inside the nominal circuit), or the (ISBCKT+1) linked list if NSBCKT is one (element inside a subckt definition). If the element is not in the list, FIND will allocate space for it and returns (a new element is read in). Otherwise, the element is already in the list, there are two possibilities: (1) The element was referenced by some other cards before (see the previous section) and nodplc(LOC+2) is set to ''.UN''; this is acceptable and the routine returns. (2) An error occurred since the program tries to redefine an element that already exists.

When called by RUNCON, IFORCE is set to 0. FIND searches for the element in the same manner as described above. When an element is not in the list, FIND will allocate space for that element and set (LOC+2)='.UN' (an element is referenced but not defined before).

When called by SUBCKT, IFORCE is set to 2. The program wants to expand the elements inside the subcircuit calls into the nominal circuit and reserve space for them accordingly (see the ERRCHK overlay).

## A.4.4. RUNCON

RUNCON determines what type of analyses to perform. It sets the variables in common blocks DC, TRAN, MISCEL, STATUS, FLAGS and OUTINF. A detailed list of these common blocks is given in [7].

## A.4.5. Storage

All input data to SLATE are maintained in the form of linked lists stored in the table IELMNT. Elements in the nominal circuit and elements inside a subcircuit definition are stored in a different manner.

Elements in the nominal circuit, device models and run condition information are stored as linked list elements. Each kind of element (e.g., R, C, and NMOS model definitions) are stored in a separate linked list identified by an ID assigned by the program (e.g., the ID of resistors is 1). The beginning of the linked lists of element type ID is pointed to by the contents of the array LOCATE(ID), which is a subscript of nodplc.

Each list entry contains a 'next pointer' that points to the next element in the list (a zero pointer denotes the end-of-the-list). Each entry also contains a VALUE subscript, usually called LOCV, that points to the first word of the real valued storage of the list element.

The elements inside a subcircuit definition are ordered into a linked list according to the size of their ID, with the element with the smallest ID located at the head of the list. The subcircuit definition forms a linked list pointed to by LOCATE(20) and each of them contains a pointer to their subcircuit elements list as shown in Fig. A.1. Note that the names of the elements inside a subcircuit are strictly local.

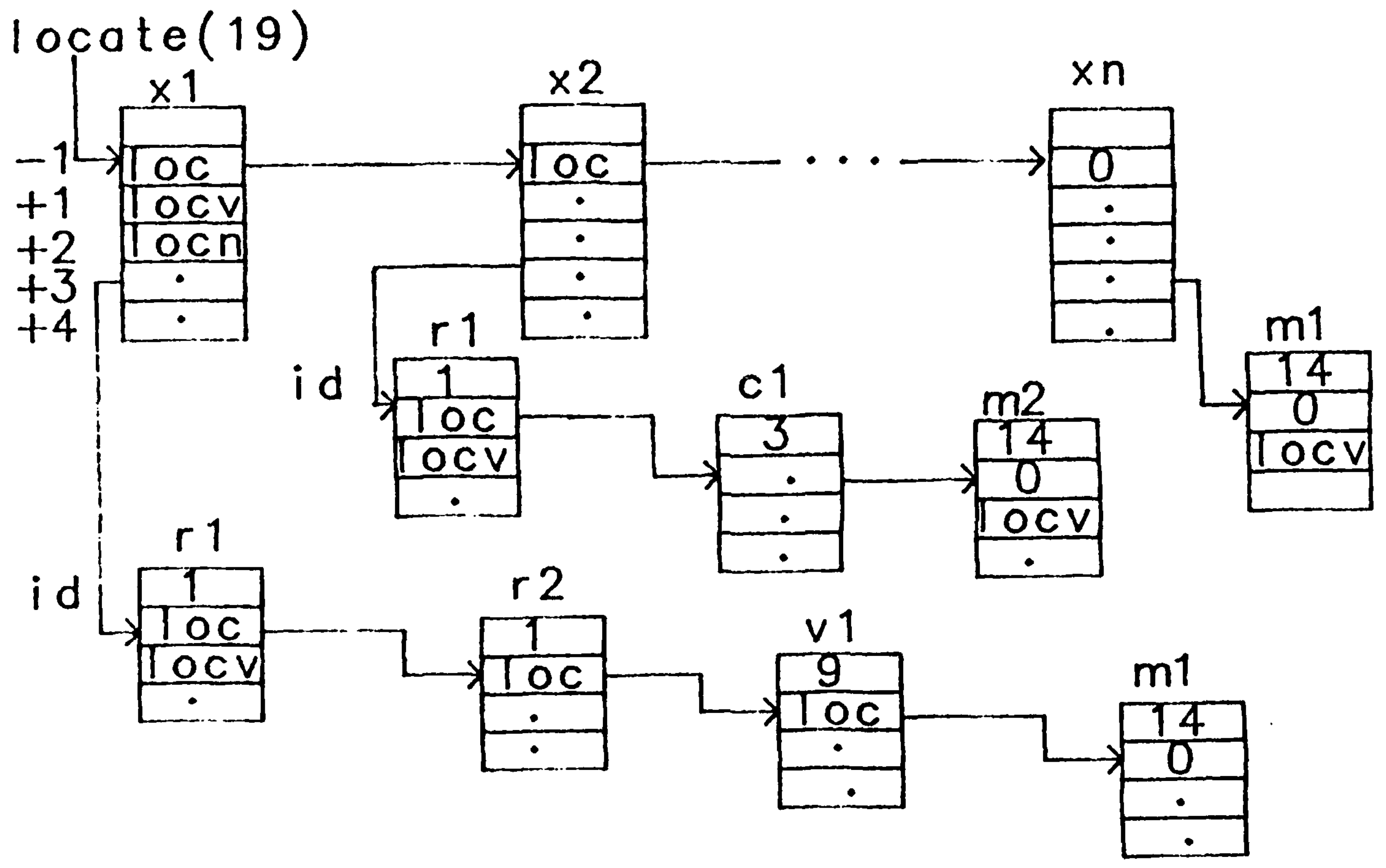


Fig. A.1 Example of the Linked List of x-elements After Subcircuit Expansion.

## A.5. ERRCHK Overlay

The ERRCHK overlay finishes the processing of input data and performs miscellaneous error checking. The overlay consists of routines ERRCHK, SHLSRT, PUTNOD, GETNOD, SUBCKT, FNDNAM, ADDELT, NEWNOD, CPYTAB, LNKREF, ELPRNT, MODCHK, TOPCHK, LINK, PUTSNO, GETSNO, NESTX and RELINK. A flowchart that describes the program flow of this overlay follows:

```
check forward references;
construct ordered list of nodes;
call SUBCKT to expand subcircuit calls;
link unsatisfied references;
set source function defaults and limits
call ELPRNT to print circuit element summary;
call TOPCHK to check topology and print node table;
invert resistance values;
change K to M for mutual inductance;
finish breakpoint table;
check analysis limits;
sequence through the output list;
store the number of elements;
relink expanded subcircuit back to subcircuit call;
call MODCHK to check and print device models;
call NESTX to process nested subcircuit calls;
call NODRES to reserve internal node for devices;
call RELINK to process subcircuit definitions;
```

### A.5.1. Forward Reference Check

After all the input data are read, all elements that are referenced by other cards should be defined. Thus the contents of nodplc(LOC+2) of all the elements should be filled by some integer data that are not equal to ".UN'. ERRCHK sequences through the element list to check that there is no element that is referenced but not defined in the circuit description.

### A.5.2. Ordered Node Tables

The nodes used in the input circuit description can be any positive integer with the ground node equals to 0 (e.g., 0,10,4,888, and 76). ERRCHK renumbers the user defined node numbers into a consecutive, compact node set starting from one. The new, ordered node numbers will be stored in place of the user defined nodes at all elements, and these new nodes will be used throughout the rest of the program.

A translation table JUNODE is constructed to give the relation between the user-defined node set and the node set used in the program. The ground node in the program is always set to 1. If the program node is i, the user's node is nodplc(JUNODE+i). For example, if the user's specified nodes are: 10, 90, 0, 45 and 80, then,

| user's node | renumbered node | JUNODE table |
|---|---|---|
| 0(ground) | 1 | nodplc(JUNODE+1)=0 |
| 10 | 2 | ( +2)=10 |
| 45 | 3 | ( +3)=45 |
| 80 | 4 | ( +4)=80 |
| 90 | 5 | ( +5)=90 |

### A.5.3. Subcircuit Expansion

ERRCHK expands out all the subcircuit calls into the nominal circuit level to check the correctness of the circuit topology and to count the number of elements and devices. The elements created from subcircuit expansion are relinked back to the corresponding subcircuit calls later in the routine.

After the READIN overlay, all the subcircuit definitions are stored in the ID=20 list and all the subcircuit calls (x-elements) are stored in the ID=19 list. The subcircuits are expanded using the algorithm given below:

```
locx=LOCATE(19); (pointer to the 1st subcircuit call)
while (locx is nonzero)
{    call FNDNAM to determine subcircuit;
     check for recursive definition of subckt calls;
     call ADDELT to add element data;
     loc=first subckt element;
     {  call FIND to add element space loce;
        (loce-1)=locx,save address of parent;
        call ADDELT to add element data;
        loc=pointer to next subckt element;
     }
     locx=pointer to next x-element;
}
```

The subcircuits are expanded in a "top-down" manner. All nested subcircuit calls (x-elements) defined within a subcircuit definition will be added into the ID=19 list and these nested x-elements will be expanded accordingly until the ID=19 list is exhausted. A list of the element pointers and variables used follows:

| ptr to element | description |
|---|---|
| locx | x-element table |
| locs | subckt definition table |
| loce | new element added |
| locv | real-valued storage |

| variables | description |
|---|---|
| NUNODS | no. of user nodes |
| NCNODS | node count |
| NUMNOD | total no. of nodes |

FNDNAM is called to determine the location of the subcircuit definition that is referenced (locs). Then SUBCKT checks for any recursive definition of subcircuit calls. The elements in the list pointed to by (locs+3) are added into the nominal circuit level using subroutines FIND and ADDELT. A dummy name dble(JELCNT(ID)) is used when calling FIND to avoid any name conflicts that might occur. Then (loce-1) of the new element is set to locx (the parent subcircuit call that invokes the adding of this element).

## A.5.4 Further Error Checking

The routines TOPCHK, ELPRNT, MODCHK, LNKREF and other error checking procedures are very similar to their corresponding parts in SPICE2; their description is not repeated here.

## A.5.5 Breakpoint Tables

In transient analysis, SLATE always uses a program calculated time step regardless of the user-specified print interval. However, the independent source waveforms frequently have sharp transitions which would cause an unnecessary reduction in the time step in order to find the exact transition time. To overcome this problem, ERRCHK generates a breakpoint table LSBKPT, which contains a sorted list of all the transition points of the independent sources. During the transient analysis, whenever the next time point is sufficiently close to the breakpoints, the time step will be adjusted so that the program lands exactly on the breakpoints.

### A.5.6. Relink

After the error checking and processing above, the elements that were expanded from the subcircuit into the nominal circuit are linked back into their subcircuit calls using the following algorithm:

```
locx=LOCATE(19); (pointer to first subckt call)
{  for (ID = 1 to 14)
    {  while (loc is nonzero)
        { if( (loc-1) equals to locx) then
            remove element from ID list;
            add element to list pointed to by (locx+4);
          endif
          loc=pointer to next element;
        }
    }
locx = pointer to next x element;
}
```

### A.5.7. Nested Subcircuit Calls

The subcircuits that contain nested subcircuit calls are processed by the routine NESTX. After subcircuit expansion, the nested subcircuit calls will be flattened out and all the x-elements will be added to the ID=19 list with the pointer to their "parent" stored in (locx-1). The routine NESTX rearranges the subcircuit calls and definitions such that the nested subcircuit calls will be transformed into one level subcircuit calls. The algorithm of NESTX is given as follows:

```
locp = LOCATE(19); (pointer to first x-element)
while (locp is nonzero)
{  locx = pointer to first element after locp;
   while (locx is nonzero)
   {if ( (locx-1) .eq. locp) then
        add nodlst of locx element to
          nodlst of locp element;
    endif
    locx = pointer to next x-element;
   }
remove redundant nodes in locp;
locp = pointer to next x-element;
}
locs = LOCATE(20); (pointer to first subckt definition)
while (locs is nonzero)
{   loce = pointer to first element;
    while (loce is nonzero)
    {   if ( loce stores an x-element) then
           add nodlst of x-element to nodlst
             of subckt definition at locs;
           remove x-element from list;
        endif
    loce = pointer to next element;
    }
locs = pointer to next subckt definition;
}
assign new node numbers for x-elements;
remove empty x-element from ID=19 list;
remove empty subckt definition from ID=20 list;
```

NESTX scans the ID=19 and ID=20 lists and determines the nesting relationships between the x-elements. The nested x-elements are torn away from their "parent" x-elements. The nodes of the parent that are connected to the nested x-element it contains are now considered as external nodes and are added to the external node list. NESTX then renumbers these new tearing nodes accordingly.

The nested x-elements that are expanded into the nominal circuit level in the ID=19 list will remain there throughout the rest of the program, while those that are in the element list of the subcircuit definitions ID=20 list are removed.

## A.5.8. Reserving Internal Nodes for Devices

The device models used in SLATE may contain internal nodes [4]. NODRES checks the device parameters and reserves additional nodes for the elements if necessary.

## A.5.9. RELINK Subroutine

The RELINK subroutine processes the subcircuits and performs further error checking. A flowchart of RELINK is given as follows:

```
locs = LOCATE(20);(pointer to first subckt definition)
while (locs is nonzero)
{       renumber nodes for subckt definition;
        store subckt information;
        locs = pointer to next subckt def.;
}
sequence through the output list;
check initial and force conditions;
```

SLATE requires a subcircuit to possess a compact and consecutive set of nodes with the external nodes ordered in the border of the BBDF matrix. RELINK renumbers the node set of each subcircuit in the

ID=20 list and stores their information:

| variables | description |
|---|---|
| NODSUB | no. of internal nodes in each subckt |
| JSTOP | no. of external nodes in each subckt |
| NXUMVS | no. of voltage defined elements in each subckt |
| NXSTOP | no. of equations in subckt =JSTOP+NODSUB+NXUMVS |
| NXMAX | maximum of all NXSTOP |
| NXTOP | sum of all NXSTOP |

The nodes of each subcircuit are renumbered and stored in place of the user defined nodes in each element in the list pointed to by (loc+4) such that:

| class of nodes | renumbered nodes |
|---|---|
| internal nodes | 1 to NODSUB (ground = 1) |
| external nodes | NODSUB+1 to NODSUB+JSTOP |

The internal nodes are renumbered from 1 to NODSUB corresponding to their size when they are defined in the user's node list; the external nodes are renumbered from NODSUB+1 to NODSUB+JSTOP corresponding to the order they appear in the subcircuit external node list. The routine then sequences through the output list and checks the initial and force conditions assigned to each subcircuit.

## A.6. SETUP Overlay

The SETUP overlay constructs the matrix structure from the circuit description. It consists of subroutines SETUP, NOFTRM, SREORD, REORDR, SREOR, REOR, MATPTR, SWAP, SSWAP and RESERV. The algorithm used in SETUP is given as follows:

```
locs = LOCATE(20);(pointer to first subckt definition)
while (locs is nonzero)
{   loc = nodplc(locs+3);(beginning of element list)
    while (loc is nonzero)
    {   call MATPTR to reserve matrix locations;
        loc =pointer to next element;
    }
    call SREORD to reorder subckt matrix;
    loc = nodplc(locs+3);(beginning of element list)
    while (loc is nonzero)
    {   store matrix locations;
        loc =pointer to next element;
    }
    locs = pointer to next subckt definition;
}
for (ID = 1 to 14) rest of the circuit
{   loc =LOCATE(ID);(pointer to first element)
    while (loc is nonzero)
    {   call MATPTR to reserve matrix locations;
        call NOFTRM to set the size of tables;
        loc = pointer to next element;
    }
}
locx = LOCATE(19);(first x-element)
while (locx is nonzero)
{   reserve matrix locations for the external
       nodes of the x-elements which form the
       border of the matrix;
    locx = pointer to next x-element;
}
call REORDR to reorder rest of the circuit;
store matrix locations for the x-elements;
store matrix locations for other elements of
   the rest of the circuit;
```

SETUP sequences through all the subcircuit definitions, calls subroutine MATPTR to build the sparse matrix structures, calls SREORD to reorder each of them, and then stores the matrix locations in the elements.

SETUP then builds the matrix structure for the rest of the circuit with the interconnections. The external nodes of the x-elements constitute the interconnection block of the equation matrix. SETUP determines which nodes in the subcircuit and the rest of the circuit are connected with the external nodes and reserves their matrix locations in the border accordingly. The program then sequences through other elements in the rest of the circuit and builds the sparse matrix structure, calls subroutine REORDR to reorder the matrix for minimal fill-in's and stores the matrix locations in each element.

## A.6.1. Matrix Structure

SETUP determines the ID of each element in the circuit and calls subroutine MATPTR to build the sparse matrix structure. Let the equation matrix be denoted by A and the elements of A be a(i,j), where i denotes the row and j the column. MATPTR uses the element stamps described in the MNA (modified nodal approach) [8] to determine which locations a(i,j) of A are being filled by introducing that element and calls RESERV to label those locations. The matrix locations are stored in two forms. In the matrix construction phase they are represented in the form of linked lists. Later, the linked list structures are transformed into sets of matrix pointer systems, which are used in the analysis part of the program.

For example, if a resistor is connected to nodes 6 and 10, then a(6,6), a(6,10), a(10,6) and a(10,10) of the equation matrix A will be filled. MATPTR introduces a new equation for each voltage defined element (i.e., L, H, E, V) since each of these elements introduces a new unknown current into the equations. (See the element stamps of [8].)

The variables and tables used in MATPTR are

| variable | description |
|---|---|
| IBR | no. of equations |
| NUMVS | no. of voltage defined elements |

| table | description |
|---|---|
| ISEQ | loc of voltage defined element |
| ISEQID | ID of voltage defined element |
| NODEVS | no. of VS connected to that node |

## A.6.2. Reserving Matrix Locations

Subroutine RESERV reserves matrix locations for the nonzero entries of the equation matrix. Suppose the program wants to reserve location (node1,node2) (i.e., row node1, column node2), the algorithm used in RESERV is

```
if (node1 or node2 is ground) return;
loc =ISR+node1;(pointer to first nonzero
       column number of row node1)
if (loc is nonzero) then
       search list for column node2;
       if (node2 is found) then
            return;
          else
```

```
            add node2 to list pointed by ISR+node1
          endif
    else
       add node2 to list pointed by ISR+node1
    endif
```

A list of the tables used in RESERV is given below:

| table | description |
|---|---|
| NUMOFF | (1 to N) no. of nonzero entries in each row<br>(N+1 to NUMSIZ) linked lists of node no. |
| NMOFFC | no. of nonzero entries in each column |
| ISR | pointers to beginning of lists in NUMOFF |
| NDIAG | set to 1 if diagonal is nonzero |

The table NUMOFF is divided into two parts: the first part of NUMOFF and the table NMOFFC store the number of nonzero entries in each row and column, respectively, and are updated whenever a matrix location is reserved. The second part of NUMOFF contains the linked lists. Each entry in the ISR table, nodplc(ISR+i), points to the beginning of the linked list which records the column numbers of nonzero locations of row i. For example, if a(3,4) and a(3,6) are nonzero, the list pointed to by nodplc(ISR+3) (row 3) should contain the nodes 4 and 6 (columns 4 and 6).

To reserve (node1,node2) the routine searches the ISR+node1 list (row node1) and determines if column node2 is in the list. If yes (node1,node2) have been reserved before and the program will return. Otherwise the location is reserved by adding node2 to the list.

## A.6.3 Compact Matrix Pointers

The linked list form structure used to set up the matrix structure has the advantage that it can be modified easily. However, this form of representation is not convenient for the analysis. Thus, SLATE generates a compact matrix pointer system after all the nonzero matrix locations are known and uses them later in the analysis.

The rest of the circuit and each of the subcircuits have their own matrix pointer tables: the matrix locations of the rest of the circuit are represented by the tables IUR, IUC, ILR and ILC; while the corresponding tables of the subcircuits are IXUR, IXUC, IXLR and IXLC. (See Section A.9.4. for the explanation of how the tables are stored.) Only the tables for the matrix locations of the rest of the circuit are described below.

The compact matrix pointer system is generated in the subroutine REOR. It divides a matrix into three parts: the matrix diagonal, the upper triangle and the lower triangle. The matrix diagonal is stored separately since it is not sparse. The upper triangle of the matrix is stored by the tables IUR (upper row) and IUC (upper column). The lower triangle is stored by the tables ILC (lower column) and ILR (lower row).

Suppose there are NSTOP equations. Then there should be NSTOP rows and NSTOP columns in the equation matrix. The IUR table contains NSTOP entries, each corresponding to a row in the upper triangle. The i'th entry of IUR, nodplc(IUR+I) points to the beginning of the part of the IUC table that stores the column numbers of the nonzero matrix locations in the upper triangle at row I.

Similarly, the ILC tables have NSTOP entries each corresponding to a column in the lower triangle, while nodplc(ILC+J) points to the beginning of the part of the ILR table that contains the row numbers of the nonzero matrix locations in the lower triangle at column J.

The variables used to describe the matrix pointers are

| <u>variable</u> | <u>description</u> |
|---|---|
| NSTOP | no. of equations |
| NUT | no. of nonzero entries in the upper triangle |
| NLT | no. of nonzero entries in the lower triangle |

## A.6.4. Storing the Matrix Locations

The values of the matrix coefficients of the "rest of the circuit" sparse matrix equations are divided into three parts and stored in the table LVN such that

| <u>entries of LVN</u> | <u>description</u> |
|---|---|
| 1 to NSTOP | stores the coefficients of the matrix diagonal. |
| NSTOP+1 to NSTOP+NUT | stores the NUT nonzero coefficients of the upper triangle, "in parallel" with IUC. |
| NSTOP+NUT+1 to NSTOP+NUT+NLT | stores the NLT nonzero coefficients of the lower triangle, "in parallel" with ILR. |

Thus, the matrix coefficients and nonzero locations can now be represented by a one-dimensional array. The actual values of the matrix coefficients will be loaded later by the subroutine YLOAD in the DCTRAN overlay. To minimize the loading time, each element in the circuit should know the positions of the matrix locations it introduces to the circuit, represented in the form of 'offsets'' of the LVN table.

The SETUP routine sequences through all the elements of the circuit. Each element in the circuit introduces several nonzero matrix locations into the equation matrix according to the MNA element stamps. For each of these matrix locations SETUP finds their equivalent offsets in the LVN table and stores them in the elements.

```
for (each element in circuit)
{   determine the matrix locations from the
        MNA element stamp;
    for (each of the matrix locations in stamp)
    {  NODEX1=row number;
       NODEX2=column number;
       if (NODEX1 or NODEX2 is 1) then
          one of the nodes is grounded,
          INDX = 1;
       else if (NODEX1.eq.NODEX2) then
          matrix location is on the diagonal,
          INDX = NODEX1;
       else if (NODEX1.lt.NODEX2) then
          matrix location in the upper triangle,
          search for column 'NODEX2'' of row
             'NODEX1'' in the IUC table and let
             its position be NS;
          INDX = NSTOP+NS;
       else
          matrix location in the lower triangle;
          search for row 'NODEX1'' of column
             'NODEX2'' in the ILR table and let
             its position be NS;
          INDX = NSTOP+NUT+NS;
       endif
       store INDX in of the element;
    }
}
```

### A.6.5 Subcircuit Reorder

SREORD partitions the subcircuit variables into three subgroups and reorders them. The algorithm used in SREORD is given as follows:

```
reorder the current variables into a subgroup;
swap the external nodes to the border;
IFLAG = 0;
for (NEXNOD = 0 to NXUMVS)
{ call SREOR,reorder the I variables;
}
for (NEXNOD = NXUMVS+1 to NSTOP)
{ if (NEXNOD > NSEND) IFLAG=1;
  call SREOR,reorder the remaining
    variables;
}
store the matrix locations in MXLOC;
```

After choosing the current variables and swapping the external nodes to the border, the variables should appear in the following order:

| position | description |
|---|---|
| 1 to NXUMVS | current variables introduced by voltage defined elements |
| NXUMVS+1 to NSEND-1 | the rest of the variables |
| NSEND-1 to NSTOP | external node voltages |

NSTOP = NSEND + NODXT

NSTOP is the sum of the number of "internal" subcircuit variables, NSEND, and external nodes, NODXT.

Each subgroup is then reordered by SREOR using the Markowitz scheme. The row swapping and node renumbering are recorded by the tables IXSWAP and IXORDR.

Finally, SREORD determines the offsets of each of the matrix locations in the matrix pointer system and stores them in the table MXLOC.

## A.6.6. Reordering the Current Variables

The algorithm used in SREORD to reorder the current variables introduced by the voltage defined elements (L, H, E, V) is given as follows [3]:

```
the external nodes are not reordered;
repeat (until all voltage sources are processed)
{   the ungrounded node of the grounded voltage
        sources are chosen first as POSITIVE,
        the voltage sources and nodes chosen
        should not be chosen again;
    whenever a node of a voltage source is
        chosen as POSITIVE, the entry in the
        NODEVS table of its negative node is
        decreased by 1;
    if the NODEVS value of a node is 1 then it
        is selected as POSITIVE. Label the
        voltage source and node so that they
        will not be chosen again;
}
swap the external nodes to the border;
```

Each of the VS (voltage defined elements) in the subcircuit is pointed to by an entry of the ISEQ table. The variable NXTV counts the number of VS processed by SREORD. In the beginning, NXTV is set to zero (i.e., no element is processed). Each time SREORD processes an VS:

- NXTV is incremented by one
- the pointer to that element in the ISEQ table

will be swapped aside so that it will not be searched again in the next pass.

- the current variable corresponding to that VS will be reordered to the top of the variable vector, according to the order that it was chosen.

- the NODEVS entry of the POSITIVE node of that element is set to 10000.

- store the direction of current flow ICPO in the chosen VS element.

The choosing process repeats until all VS are processed (i.e., NXTV=NUMVS-1).

The table NODEVS contains the number of voltage defined elements connected to each node. This determines the order in which a node is chosen as POSITIVE. The ungrounded node of the grounded VS are chosen first, followed in turn by nodes with the least number of number of VS connected to it. (The nodes with a smaller number of VS connected to it have less off-diagonal terms in the matrix equations.) To prevent any node from being chosen, NODEVS+node is set to 10000. (This represents a very low priority in the reordering scheme.)

The direction of current flow in a VS is represented by ICPO. Its value is one if current is flowing from the positive terminal of the device to its negative terminal, otherwise ICPO=-1.

### A.6.7 Rest of the Circuit Reorder

The equation matrix of the rest of the circuit is reordered by the subroutine REORDR. The reordering scheme is almost the same as the one used in subcircuit reordering. The current variables are further divided as (1) $I_1$ and $I_2$, (2) $V_v$, and (3) the remaining variables. $I_1$ is partitioned into groups of $I_v$, $I_{vcv}$ and $I_{ccv}$, which are the currents introduced by independent, voltage controlled and current controlled voltage sources, respectively.

After partitioning and processing of the current variables using the scheme described in the previous section, each group of variables is reordered using the Markowitz scheme by the subroutine REOR. The offsets of the matrix locations in the matrix pointer tables are then computed and stored in the table MLOC.

## A.7. DCTRAN Overlay

The DCTRAN overlay performs the dc transfer curve, dc operating point, initial transient operating point and transient analyses. The overlay consists of routines DCTRAN, DCDCM, COMCOF, ITER8, TRUNC, SORUPD, YLOAD, NTRPL8, EVTERM, NXTPWR, INTGR8, DIODE, BJT, JFET, MOSFET, and MOSFEQ1.

The types of analysis to be performed are determined in the overlay root and indicated by the flags described as follows:

| flag | value | meaning |
|---|---|---|
| MODE | 1 | dc analysis (subtype defined by MODEDC) |
|  | 2 | transient analysis |
| MODEDC | 1 | dc operating point |
|  | 2 | initial operating point for transient analysis |
|  | 3 | dc transfer curve computation |
| INITF | 1 | converge with 'off' devices allowed to float |
|  | 2 | initialize junction voltages |
|  | 3 | converge with 'off' parameter held 'off' |
|  | 4 | <unused> |
|  | 5 | first time point in transient analysis |
|  | 6 | prediction step |
| IGOOF | 0 | converged |
|  | other | not converged |

### A.7.1. DC Operating Point

If both values of MODE and MODEDC are 1, the dc operating point is computed. A flowchart of this is given as follows:

```
initialize;
```

```
TIME = 0.0;
call SORUPD to set sources to time zero values;
INITF = 2;
call ITER8;
print operating point;
```

The actual sparse matrix equation solution is carried out in the subroutine ITER8. The value of INITF is set to 2 to initialize the junction voltages.

### A.7.2. Transient Initial Conditions

If the values of MODE and MODEDC are 1 and 2, respectively, the DCTRAN overlay will compute a set of initial circuit conditions prior to the transient analysis. A flowchart for the initial transient solution is given as follows:

```
initialize;
TIME = 0.0;
call SORUPD to set sources to time zero values;
INITF = 2;
call ITER8;
if (converged)
    { print solution;}
```

### A.7.3. DC Transfer Curve

The dc transfer curve is simply a repetitive dc operating point computation performed for a range of values for one independent voltage or current source in the circuit. If the values of MODE and MODEDC are 1 and 3, respectively, the dc transfer curve is computed.

A flowchart of this is given as follows:

```
initialize; TIME = 0.0;
call SORUPD to set sources to time 0 values;
INITF = 2;(initialize junction voltages)
set all subckts to be nonlatent;
for (each source value)
{    if (INITF .ne. 2) INITF = 6;
   { call ITER8;
     if (not converged) stop analysis;
     locx = locate(19);(first subckt)
     while (locx is nonzero)
     {  check for the latency condition;
        if (subckt is latent) then
           nodplc(locx+9) = 1;
        else
           nodplc(locx+9) = 0;
        endif
     }
     store outputs;
   }
```

All subcircuits are assumed to be nonlatent in the beginning. During iteration, the program checks all the subcircuits for latency at each of the source values using the conditions of equation (3.4). The voltage values of the present and previous iteration points are stored in tables LVIM1 and LDO, respectively.

The analysis of subcircuit $N_k$ will be omitted in the next iteration point and afterwards until it is determined to be nonlatent again.

After the first sweep point, the value of INITF is set to 6. The piecewise nonlinear method [3] is used to predict the solution at the next sweep point and used as the initial guess at the next iteration.

## A.7.4. Transient Analysis

The transient analysis is performed if MODE=2. A flowchart of the transient analysis is given as follows:

```
          initialize; TIME=0.0; DELTA=TSTEP;
          INITF = 5;
savout:   store outputs in LOUTPT table;
newtim:   TIME = TIME+DELTA;
          if (TIME > TSTOP) exit;
          { adjust DELTA for breakpoint table values;
            call SORUPD;
            call ITER8;
          }
          if (converged) goto tsterr;
          { TIME = TIME-DELTA;
            DELTA = DELTA/8;
            goto tsdel;
          }
tsterr:   locx = LOCATE(19); (first subckt)
          while (locx is nonzero)
          { if (all external nodes are latent) then
                 {if (nodplc(locx+9).ne.1) then
                       nodplc(locx+9) = -1;
                  endif
                 }
            else
                 nodplc(locx+9) = 0;(nonlatent)
            endif
            locx = nodplc(locx);
          }
          call TRUNC;
          if (error acceptable) goto savout;
          { TIME = TIME-DELTA;
            DELTA = DELNEW (computed in TRUNC);
          }
tsdel:    if (DELTA < DELMIN) stop analysis;
          goto newtim;
```

There are four different latency conditions possible for a sub-circuit:

| (locx+9) | condition of subckt |
|---|---|
| 0 | not latent |
| 1 | latent at the time level |

| | |
|---|---|
| | (determined in TRUNC and DCTRAN) |
| -1 | latent in Newton-Raphson iteration (determined in ITER8) |
| -2 | all energy storage elements are latent (determined in YLOAD) |

The scheme 2 proposed in [3] is used to determine the latency in time. After each iteration time point, DCTRAN checks the external node voltages of each of the subcircuits: (1) If the changes of all the external node voltages between the previous and present time points are less than the tolerance and the subcircuit is originally latent in time, it remains latent in time (see equations (3.3) and (3.9)). (2) If all the external nodes are latent but the subcircuit is not latent in time, it is declared to be latent in the N-R iteration. (3) If the external node voltages are not latent, the subcircuit is declared to be nonlatent.

## A.7.5. Determining the New Time Step

The estimation of the new time step to be used is performed in the subroutine TRUNC. The algorithm used in TRUNC is given below:

```
DELNEW = TSTOP;
for (each of the energy storage elements)
{   branch to TERR; }
locx = LOCATE(19)
while ( locx is nonzero)
{ if (subckt latent in time) goto nxckt;
  INEED2 = 0;
  if (subckt latent in N-R iteration) INEED2=1;
   for each (energy storage element in subckt)
   { branch to TERR;
     }
  if (subckt latent in N-R iteration) then
     nodplc(locx+9) = INEED2;
```

```
      endif
    nxckt: locx = nodplc(locx)
    }
    TERR: find current and charge error tolerance;
          estimate new time step DEL for this branch;
          if (element is not slowly varying) INEED2=0;
          DELNEW = min(DELNEW,DEL);
```

The program checks if all the energy storage devices inside the subcircuits are slowly varying (equation (3.8)) and sets the value of nodplc(locx+9) accordingly.

## A.7.6. The Iteration Scheme in SLATE

The actual Newton Raphson iteration is controlled by the subroutine ITERS. The algorithm used in the subroutine is listed below:

```
    ITERNO = NONCON =0;
    done = .false.
    while (not done)
    {   call YLOAD;
        if ((NOSOLV is nonzero) and
            (analysis is initial transient)) exit;
        ITERNO = ITERNO+1;
        switch (INITF) of
           { "1": if (NONCON=0) exit;
                  goto solve;
             "2": INITF=3; goto solve;
             "3": if (NONCON=0) INITF=1;
                  goto solve;
             "4","5","6": INITF=1;
           }
    solve:
        if (ITERNO>iteration limit) exit;
        if (IFINI is nonzero) force node voltages;
        call DCDCMP;
        call DCSOL;
        NONTMP = NONCON;
        NONCON = 0;(.done.)
        if (NONTMP=0 and not 1st iteration) then
            if (NOT ALL circuit node voltages
                converged) NONCON=NONCON+1;
        endif
        locx = LOCATE(19); (first subckt)
        while (locx is nonzero)
        {   if (all external nodes converged) then
                 if (nodplc(locx+9)= 1 or -1) goto nxtckt;
                 if (nodplc(locx+9)=-2) nodplc(locx+9)=-1;
              else
                 nodplc(locx+9)=0;
              endif
            goto sdcsol;
            if (NOT ALL internal node converged) then
               NONCON =NONCON+1;
               nodplc(locx+9) = 0;
            endif
    nxtckt: locx = pointer to next subckt;
        }
    }
    sdcsol: back substitution to solve subckt equation;
```

ITER8 first calls YLOAD to load the equation matrix and decompose the subcircuit matrices. It then calls DCDCMP to LU factorize the rest of the circuit and interconnection matrix and calls DCSOL to solve them (steps 3, 4 and 5 of algorithm $S_1$). ITER8 then checks and updates the latency of the subcircuits and uses SDCSOL to obtain the solution of each of the subcircuit blocks (step 6 of algorithm $S_1$).

Some subcircuits might converge in fewer iterations than the others. They can be declared to be latent at the Newton-Raphson level (nodplc(locx+9)=-1) if (1) nodplc(locx+9)=-2 (determined in YLOAD), and (2) all external and internal node voltages have converged. The loading, LU factorizing, forward and backward substitutions of these latent subcircuits will be omitted until they are declared to be nonlatent again.

If the program wants to force the node voltages (IFINI=1), it will store the desired values of the variables in their positions in the solution table LVN and set their corresponding entries in the NNDIAG table to -1. The LU factorization, forward and backward substitution of the rows and columns corresponding to that node will be omitted.

## A.7.7. Element Load

The subroutine YLOAD loads the equation matrix, decomposes the subcircuit matrices and carries out steps 1 and 2 of the equation solution procedure $S_1$. The algorithm used in the subroutine is given below:

```
LATENT = 0;(assume elements are latent)
for (id= 1 to 14)
{  load elements in rest of the circuit;
}
if (LATENT is nonzero) NONCON=NONCON+1;
locx = LOCATE(19);
while (locx is nonzero)
{      LATENT = 0;
       if (subckt is latent) goto nsbckt;
           LATENT = 0;
           load elements in the subckt;
           if (LATENT = 0) then
               nodplc(locx+9) = -2;(all elements latent)
           else
               NONCON = NONCON+1;
           endif
nsbckt: locx = nodplc(locx);
}
if (INITF = 2 or 3) NONCON=1;
locx = LOCATE(19);
while (locx is nonzero)
{      decompose subckt submatrices, perform
           forward substitution, goto sdcdcm;
       solve for y = W^T a;
}
```

The matrix coefficients are loaded one element at a time. A concise description of how each element type is loaded can be found in [7]. The rest of the circuit is loaded first, followed by the subcircuits.

In order for a circuit or subcircuit solution to converge, all energy storage elements and controlled sources must remain latent (i.e., LATENT=0). The changes of their values between the last two iterations must be less than the tolerance TOL (equation (3.8)).

After loading, the matrices of the subcircuits are LU factorized and forward substitution is carried out (step 1 of algorithm $S_1$ of [3]) to solve for a, followed by step 2 to find the vector y. The results (stored in the table LXVN) are subtracted from the source vector $J_{tsk}$ (the right-hand side of the equation in step 3 of $S_1$) and stored in the table LVN.

## A.8. Linked List Specifications

The linked list elements used in SLATE are essentially the same as those used in SPICE2 [7]. Only the specifications of the subcircuit calls and definitions are given. All integer data referred to are stored in the array NODPLC; all real and character data values are stored in the array VALUE. The NODPLC subscript of the linked list element is called LOC, while NODPLC(LOC+1) stores the pointer LOCV, which is a subscript of VALUE that contains the real valued storage of that element.

## A.8.1. Subcircuit Call

ID = 19

```
     - 1: subckt info
LOCX+ 0: next pointer
     + 1: LOCV                           LOCV + 0: element name
     + 2: tp(external nodes)
     + 3: tp( subckt definition)
     + 4: tp(element list)
     + 5: NOFFSV
     + 6: NODXT
     + 7: LXNOD
     + 8: NXTOP
     + 9: latency flag
     +10: NDIST
     +11: tp(nodes and node voltages forced)
     +12: size of forced node table at (locx+11)
     +13: tp(nodes and node voltages initialized)
     +14: size of initialized node table at (locx+13)
```

Comments:

(1) nodplc(locx+4) points to a linked list of elements expanded from the subckt definitions (see the ERRCHK overlay).

(2) nodplc(locx+7) points to the table LXNOD which stores the matrix locations of the bordered block created by the tearing nodes.

(3) nodplc(locx+9) indicates the latency condition:

| | |
|---|---|
| 1 | latent in time |
| 0 | not latent |
| -1 | converged in N-R iteration |
| -2 | all energy storage devices and controlled sources converged |

(4) LDO+NDIST is the beginning of the table that stores the subckt tearing node voltages.

(5) The forced node table contains the list of nodes to be forced and their voltage values. It is constructed when a .FORCE card is read with the subckt's name on it. If

LOCT = nodplc(locx+11) = ptr to forced node table, and

NTMP = nodplc(locx+12) = size of LOCT table

then

```
LOCT+      0: NODE_1
    +      1: V_1          NODE_i = node to be forced
    +      2: NODE_2
    +      3: v_2          V_i    = forced node voltage
  :
  :
    +NTMP-2: NODE_k
    +NTMP-1: V_k
```

(6) The initialized node table contains the list of nodes to be initialized and their values. It is constructed when a .INITIAL card is read with the subckt's name on it. The data are stored in the same manner as in the force node table.

## A.8.2. Subcircuit Definition

```
ID = 20

    - 1: subckt info
LOCS+ 0: next pointer
    + 1: LOCV                              LOCV + 0: element name
    + 2: tp(table of external nodes)
    + 3: tp(element linked list)
    + 4: NXSTOP
    + 5: NXUMVS
    + 6: NOFFSW
    + 7: NOFFUC
    + 8: NOFFLR
    + 9: NOFFML
    +10: NSEND
    +11: size of the LXVN table
    +12: NUT
    +13: tp(old external node list)
```

Comments:

(7) Size of LXVN table = no. of nonzero matrix locations + no. of equations = NUT+NLT+NSTOP+NSTOP.

(8) The user's defined internal nodes are stored in a table pointed to by nodplc(locs+13).

## A.9. Labeled COMMON Blocks

The COMMON blocks that are used in SLATE only are listed below.

### A.9.1. CY

The COMMON block CY contains the tables and variables used in setting up and reordering the matrix structure of the nominal circuit.

| name | description |
|---|---|
| NUMVS2 | no. of independent voltage sources |
| NSTOP1 | ptr to the beginning of the $I_2$ equations |
| NSTOP2 | ptr to the beginning of the V equations |

The other variables can be found in the SPICE2 report [7]; they are not listed again.

### A.9.2. CP1

The COMMON block contains the tables that store the tables used in subckt reordering. The beginnings of the tables in CP3 are given by adding offset values, contained in CP2, to their corresponding tables in this block.

| name | description |
|---|---|
| ISSWAP | IXSWAP=ISSWAP+NOFFSW |
| ISORDR | IXORDR=ISORDR+NOFFSW |
| ISUR | IXUR=ISUR+NOFFSW |
| ISUC | IXUC=ISUC+NOFFUC |
| ISLR | IXLR=ISLR+NOFFLR |
| MSLOC | MXLOC=MSLOC+NOFFML |

### A.9.3. CP2

This COMMON block contains the offsets of the subcircuit tables from the tables used in the COMMON block CP1.

| name | description |
|---|---|
| NO6 | <unused> |
| NO8 | " |
| NO4 | " |
| NOFFSW | <see COMMON block CP1> |
| NOFFUC | " " |
| NOFFLR | " " |
| NOFFML | " " |

### A.9.4. CP3

This COMMON block contains the subcircuit tables used in SETUP and the subckt matrix pointer tables.

| name | description |
|---|---|
| IXSWAP | tp(record of equation swaps) |
| IXORDR | tp(record of equation reorder) |
| IXUR | tp(IXUC indices) |
| IXLC | tp(IXLR indices) |
| IXUC | tp(nonzero columns in upper triangle) |
| IXLR | tp(nonzero rows in lower triangle) |
| MXLOC | tp(compact matrix pointers) |

## A.9.5. CP4

This COMMON block contains the individual subcircuit matrix pointer tables and the tables used in the matrix setup phase.

| name | description |
|---|---|
| NOFFSV | offset of LVN from the top of LSVN, stored in nodplc(locx+5) |
| NXTOP | offset of LXVIM1 from the top of LSVIM1, stored in nodplc(locx+8) |
| NXMAX | the maximum no. of equations in one subckt |

## A.9.6. CP5

This COMMON block contains the tp that stores the subckt matrix coefficients.

| name | description |
|---|---|
| LXVN | tp(right-hand-side of subckt equations) |
| LXYNL | LXVN offset: matrix diagonal terms |
| LXYU | LXVN offset: matrix upper triangular terms |
| LXYL | LXVN offset: matrix lower triangular terms |
| LXVIM1 | tp(previous subckt solution, copy of LXVN) |
| NXSTOP | no. of equations in the subckt |

## A.9.7. CP6

This COMMON block contains the tables that store all the double precision tables contained in the COMMON block CP5.

| name | description |
|---|---|
| LLSVIM | tp(used to create LSVIM1) |
| LLSVN | tp(used to create LSVN) |
| LSVIM1 | LXVIM1=LSVIM1+NXTOP |
| LSVN | LXVN=LSVN+NOFFSV |

### A.9.8. CP7

This COMMON block contains the variables used in subcircuit equation reordering.

| name | description |
|---|---|
| NXUMVS | no. of independent voltage sources |
| NXUVS2 | <unused> |
| NXTOP1 | ptr to the beginning of the V equations |
| NXTOP2 | <unused> |
| NSEND | = NSTOP-NODXT = no. of subckt equations - no. of tearing node equations |

### A.9.9. CP8

This COMMON block is used in subroutine YLOAD.

| name | description |
|---|---|
| LATENT | =0 => all energy storage devices are latent |

### A.9.10. CP9

This COMMON block contains the subcircuit latency statistics.

| name | description |
|---|---|
| ITOTAL | total no. of subckt times the total no. of iterations |
| ILATNT | no. of nonlatent subckt times the no. of iterations for those subckt |
| NNDIAG | tp(nodes to be forced or initialized) (NNDIAG+node)=-1 => LU factorization and back substitution of the node should be skipped. (LXVN+node) contains the forced solution |

## A.9.11 EXT

This COMMON block contains the variables that record the size of the tables contained in COMMON block CP1.

| name | description |
|---|---|
| NMXEXT | size of MSLOC table |
| NUCEXT | size of ISUC table |
| NLREXT | size of ISLR table |
| NUMEXT | size of NUMOFF table |
| NUMSIZ | ptr to the list of reserved nodes in the table NUMOFF (used in RESERV) |

## A.9.12. FORCE

This COMMON block contains the variables used in forcing or initializing the node voltages.

| name | description |
|---|---|
| NOFOR | no. of nodes to be forced |
| IVFOR | tp(values of forced node voltages) |
| INFOR | tp(nodes to be forced) |
| NOINI | no. of node to be initialized |
| IVINI | tp(values of the initialized node voltages) |
| ININI | tp(node no. to be initialized) |
| IFINI | =1 => nodes have to be initialized |

Example A.1

This is an example of a 4 bit full adder implemented using nested subcircuit calls:

```
* Four Bit Full Adder Using Nested Subcircuits
*  two input nand gate: input(2) output vdd
.subckt nand 1 2 3 4
m1 3 1 4 4 cmosp l=3 w=14 ad=63 as=63 asd=42 ass=42 rdd=35 rss=35
m2 3 2 4 4 cmosp l=3 w=14 ad=63 as=63  asd=42 ass=42 rdd=35 rss=35
m3 3 1 5 0 cmosn l=3 w=7  ad=49 as=25  asd=28 ass=14 rdd=35 rss=35
m4 5 2 0 0 cmosn l=3 w=7  ad=25 as=49  asd=14 ass=28 rdd=35 rss=35
.ends
*  inverter : input, output, vdd
.subckt inv 1 2 3
m1 2 1 3 3 cmosp l=3 w=14 ad=63 as=63  asd=42 ass=42 rdd=35 rss=35
m2 2 1 0 0 cmosn l=3 w=7  ad=49 as=49  asd=28 ass=28 rdd=35 rss=35
.ends
* exclusive or gate
.subckt xor 10 20 30 40
x1 10 50 40 inv
x2 20 60 40 inv
x3 10 60 70 40 nand
x4 20 50 80 40 nand
x5 70 80 30 40 nand
.ends
* 1 bit full adder: input(2), cin, cout, su m vdd
.subckt adder 1 2 3 4 5 6
x1  1 2 7 6 xor
x2  7 3 5 6 xor
*
x3  7 3 8 6 nand
x4  1 2 9 6 nand
x5  8 9 4 6 nand
.ends
* All input bits changes fro m 0 volts to 5 Volts
va  1 0 pulse (0 5 0n 2n 2n 100n 1000n)
vdd 6 0 dc 5v
x1  1 1 1 5 6 adder
x2  1 1 4 9 10 6 adder
x3  1 1 9 13 14 6 adder
x4  1 1 13 17 18 6 adder
.model cmosp pmos vto=-1.1 n=5e16 kp=8u cox=.345f lambda=.025
.+   be=.52  ms=.33 kpn=.0918f lgos=.4 lgod=.4 tld=1.0
.model cmosn nmos vto=1.1 n=1e16 kp=22u cox=.345f lambda=.052
  be=.52  ms=.33 kpn=.0918f lgos=.4 lgod=.4 tld=1.0
.options  nolist noopts no mod nonode
```

```
.tran 1n 80n
.print tran v(1) v(5) v(9) v(13) v(17) v(18)
.end
```

## REFERENCES

[1] L. W. Nagel,"SPICE2: A Computer Program to Simulate Semiconductor Circuits," Electron. Res. Lab., University of California Berkeley, ERL Memo No. ERL-M250, May 1975.

[2] I. Hajj, P. Yang and T.N. Trick,"Avoiding Zero Pivots in the Modified Nodal Approach", IEEE Transaction on Circuit and Systems , vol. CAS-28. No. 4, pp. 271-279, April 1981.

[3] P. Yang, "An Investigation of Ordering, Tearing and Latency Algorithms for the Time-Domain Solution of Large Circuits," Ph.D. Thesis, University of Illinois at Urbana-Champaign, August 1980.

[4] S. A. Sohail, "A Simulation Program with Latency Exploitation for the Transient Analysis of Digital Circuits," M.S. Thesis, University of Illinois at Urbana-Champaign, August 1983.

[5] L. O. Chua and L. K. Chen, "Diakoptik and Generalized Hybrid Analysis," IEEE Transaction Circuits and Systems, vol. CAS-23, No. 12, pp. 694-705, Dec. 1976.

[6] I. Hajj, "Sparsity Considerations in Network Solution by Tearing", IEEE Transaction Circuits and Systems, vol. CAS-27, No. 5, pp. 357-366, May 1980.

[7] E. Cohen, "Program Reference Guide for SPICE2," Report No. ERL-M592, University of California, Berkeley, June 1976.

[8] C. W. Ho, A. E. Reuhli, P. A. Brennan, "The Modified Nodal Approach to Network Analysis," Proc. 1974 IEEE International Symposium on Circuits and Systems , San Francisco, pp. 505-509, April 1974.